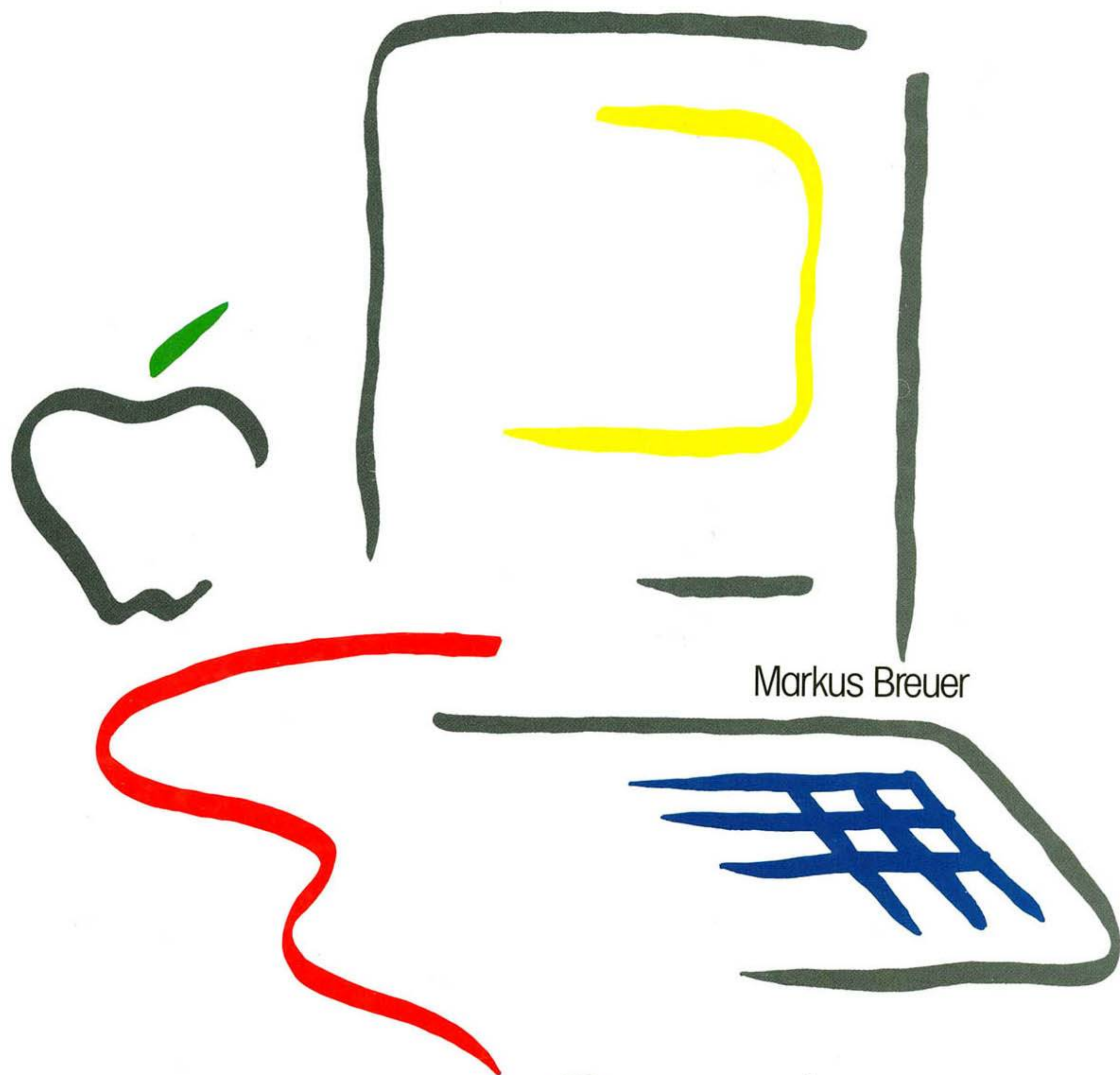
Markt&Technik



Markus Breuer

Die Programmierung des Macintosh

Einführung in die Programmerstellung mit der Macintosh-ToolBox in Pascal

Die Programmierung des Macintosh

Markus Breuer

Die Programmierung des Macintosh

Einführung in die Programmerstellung mit der Macintosh-ToolBox in Pascal

Markt&Technik Verlag

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Breuer, Markus:
Die Programmierung des Macintosh : Einf. in d. Programmerstellung
mit d. Macintosh-ToolBox in Pascal / Markus Breuer. -
Haar bei München : Markt-und-Technik-Verlag, 1986. –
ISBN 3-89090-184-0

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
89 88 87 86

ISBN 3-89090-184-0

Einbandgestaltung: Grafikdesign Heinz Rauner
Druck: Jantsch, Günzburg
Printed in Germany

Inhaltsverzeichnis

Vorwort

Der Macintosh ist anders als andere Computer! Er hat den Weg bereitet für eine ganze Generation von neuen MicroComputern. Diese neue Generation kommt jetzt nach und nach auf den Markt und hat vor allen Dingen eins gemeinsam. Nicht etwa die Maus und die Fenster! Nein, alle diese Rechner sind in ihrer Bedienung ganz unvergleichlich viel leichter und leichter zu erlernen als alle Computer vor ihnen. Für den Computer-Anwender ist der Macintosh ein Meilenstein (hoffentlich nicht der letzte) auf dem Weg zum Computer für jedermann.

Für den Programmierer sieht das schon etwas anders aus! Einerseits bietet ihm der Macintosh auch vollkommen neue Möglichkeiten, seine Programme zu gestalten und mit dem Anwender zu kommunizieren. Auch die Programmentwicklung auf dem Macintosh nutzt zum Teil die vielen Eigenschaften, die ihn aus der Masse der Computer hervorheben. Viele der Programmierwerkzeuge wie Editoren und Compiler werden dadurch ebenfalls leichter zu bedienen als auf anderen Rechnern.

Andererseits sind viele der Konzepte des Macintosh — gerade durch ihre Neuartigkeit — für den Programmierer schwer zu bewältigen. Erfahrungen mit anderen Computern sind oft sogar hinderlich, da man liebgewordene Gewohnheiten ablegen muß, um mit den Regeln der Macintosh-Programmierung fertig zu werden.

Dieses Buch wendet sich an diejenigen, die die erste Phase des Staunens über die Eigenschaften des Macs, wie ihn seine Fans liebevoll nennen, hinter sich haben und nun auch selbst Programme entwickeln wollen, die diese Eigenschaften voll ausnutzen: die Maus, Fenster, Menüs, die auf Knopfdruck herunterklappen, und die vielen grafischen Möglichkeiten. Nicht jeder Besitzer des Macintosh wird selbst Programme entwickeln wollen. Aber die, die den Ehrgeiz besitzen, werden rasch feststellen, daß es nicht gerade leicht ist, diese vielfältigen Möglichkeiten wirklich zu beherrschen. Am Anfang erscheinen oft die trivialsten Barrieren unüberwindlich. Dies gilt nicht nur für Anfänger auf dem Gebiet der Programmierung, sondern auch für alle, die schon auf anderen Computern Kontakt mit einer Programmiersprache geschlossen haben. Ich selbst war da keine Ausnahme.

Als Informatiker habe ich zwar schon mit vielen Computern und Programmiersprachen gearbeitet. Trotzdem hat es Monate gedauert, bis mir der Macintosh vertraut genug war, um vom Herumprobieren zum ernsthaften Arbeiten überzugehen. Und dies, obwohl mir ein Kollege, der schon etwas länger am Macintosh gearbeitet hatte, mit Rat und Tat zur Seite stand. Der Hauptgrund für diese Schwierigkeiten liegt natürlich in der Neuartigkeit und damit auch Fremdartigkeit vieler Konzepte des Macintosh. Um die Programmierung zu vereinfachen, hat Apple zwar bereits eine Fülle hilfreicher Unterprogramme fest in den Macintosh eingebaut; die sog. **Toolbox** (zu deutsch: 'Werkzeugkiste') im ROM des Mac. Aber gerade diese Fülle ist am Anfang schwer durchschaubar und macht die Einarbeitung nur um so schwieriger. Was mir fehlte, und gewiß auch den meisten andern Einsteigern in die Programmierung des Macintosh fehlen wird, ist eine Einführung, die sich auf das Wesentliche beschränkt. Die zunächst die wichtigsten Konzepte vorstellt und überflüssige Details wegläßt oder erst nach und nach einführt.

Genau das will das vorliegende Werk leisten! Um dem Leser eine ungefähre Vorstellung davon zu geben, was er in folgenden Kapiteln zu erwarten hat, möchte ich nun erst einmal ein paar Worte darüber verlieren, was dieses Buch ist, was es nicht ist, wie es aufgebaut ist und welche Erwartungen ich an den Leser stelle.

Was dieses Buch sein will:

Die folgenden Kapitel wollen eine Einführung in die Programmierung des Macintosh und vielleicht auch anderer Computer mit einer vergleichbaren Bediener-Oberfläche sein. Anhand eines großen Beispiel-Programms, das alle wichtigen Eigenschaften des Macintosh zeigt – und vieler kleinerer Beispiele – werden die wichtigsten Prinzipien und Techniken, die für die Programmierung des Macintosh benötigt werden, vorgestellt. Im Vordergrund steht dabei die geschickte und korrekte Benutzung der vielen Hilfs-Routinen im ROM des Macintosh, der sog. **ToolBox**. Am Ende wird der Leser fähig sein, auch selber eigene Programm-Ideen "mac-like" zu verwirklichen.

Um dies zu erreichen, war es vor allem notwendig, eine Auswahl aus der Vielzahl der im ROM des Macintosh liegenden Hilfsroutinen zu treffen. Ich hoffe, daß es mir gelungen ist, das Wesentliche und Notwendige in dieses Buch aufzunehmen und alle überflüssigen Details fortzulassen. Der Leser möge nicht enttäuscht sein, wenn er nicht zu jedem Aspekt des Macintosh, der ihn gerade interessiert, etwas auf den folgenden Seiten finden wird.

Einige Teilgebiete der ToolBox und des Betriebssystems im ROM des Macintosh decke ich überhaupt nicht ab, und auch die Beschreibung der aufgenommenen Teilgebiete erhebt keinen Anspruch auf Vollständigkeit.

Dieses Buch ist im wesentlichen ein Lehrbuch, kein Nachschlagewerk. Die Reihenfolge der einzelnen Kapitel wird durch die Bedürfnisse der vorgestellten Beispielprogramme bestimmt, soweit dies möglich ist.

Die Beispielprogramme wie auch alle Beschreibungen von Prozeduren und Datenstrukturen orientieren sich an der Programmiersprache Pascal. Sie dürften jedoch auch für Benutzer anderer Sprachen verwendbar sein. Ich selbst habe die meisten der Beispielprogramme sowohl in Pascal wie auch in Modula-2 und C ausprobiert. Die Übertragung von Pascal in eine dieser Sprachen ging stets schnell und nach einiger Übung relativ problemlos, weswegen ich annehme, daß Benutzer dieser 3 Sprachen keine Schwierigkeiten haben werden, Nutzen aus diesem Buch zu ziehen. Für Anwender von Sprachen, die kaum Ähnlichkeit mit Pascal haben (wie BASIC, Forth und Logo), werden nur die Kapitel, die sich mit der Beschreibung von Teilaspekten der ToolBox beschäftigen interessant sein. Die Beispielprogramme jedoch werden nur mit umfangreichen Änderungen oder gar nicht verwendbar sein.

Was dieses Buch nicht sein kann:

Obwohl es vielleicht ganz wünschenswert gewesen wäre, war es mir und dem Verlag nicht möglich, neben der hier gebotenen Einführung eine komplette deutsche Übersetzung des Programmierhandbuchs *Inside Macintosh* mit aufzunehmen (aus Platzgründen unmöglich) oder parallel anzubieten. In der vorliegenden Form des Buches kann *Inside Macintosh* deshalb nicht ersetzt, sondern nur ergänzt werden. Auch die Kapitel, in denen einzelne Teile der ToolBox relativ ausführlich erläutert werden, decken nur einen — manchmal kleinen — Teil der entsprechenden Abschnitte in *Inside Macintosh* ab. Deshalb kann jedem, der sich ernsthaft mit der Programmierung des Macintosh beschäftigen will, die Anschaffung von *Inside Macintosh* (im folgenden oft kurz *IM* genannt) nur wärmstens empfohlen werden.

Genausowenig, wie es das echte Programmierhandbuch des Macintosh ersetzen kann, kann dies ein Lehrbuch der Programmierung im allgemeinen oder einer bestimmten Programmiersprache im speziellen sein. Eigentlich jedes Kapitel geht von einem Grundwissen zum Thema Programmierung aus, das hier nicht vermittelt werden kann.

Und schließlich enthalten die folgenden Kapitel auch kein Handbuch einer bestimmten Programmiersprache. Für den Macintosh sind inzwischen auch so viele gute Compiler und Interpreter für die verschiedensten Sprachen auf dem Markt, daß es zu schwer gewesen wäre, sich für ein System zu entscheiden. Dies ändert sich vielleicht in einer folgenden Auflage, falls Apple selbst bis dahin eine höhere Programmiersprache auf dem Macintosh anbietet, mit der selbständig arbeitende Programme geschrieben werden können.

Apple hat eine solche Programmentwicklungs-Umgebung zwar bereits angekündigt; Details waren aber bis zur Konzeptionierung dieses Buches (Frühjahr/Sommer '85) nicht zu erfahren. Ich verwende deshalb für alle Programmbeispiele einen Dialekt der Sprache Pascal, auf dem auch die Erläuterungen in IM basieren: LisaPascal. Das angekündigte Programmiersystem für den Macintosh soll auch einen Pascal-Compiler enthalten, der kompatibel zu LisaPascal ist.

Zum Aufbau des Buches:

Die folgenden Kapitel gliedern sich in zwei Teile. Im ersten Teil versuche ich, Grundlagenwissen zu vermitteln. Dieser Teil wird vielen Lesern wahrscheinlich etwas trocken vorkommen, da er, bis auf eine Ausnahme, kein komplettes Beispielprogramm enthält. Dieser erste Teil ist aber trotzdem unvermeidlich, da selbst das simpelste Programm, das nur einige der Eigenschaften nutzt, die den Macintosh aus der Masse der Computer hervorheben, bereits all die Teile der ToolBox benutzt, die der erste Teil erläutert.

Nach dieser "Durststrecke" wird im zweiten Teil das *Rahmenprogramm* vorgestellt und ausgebaut. Das Rahmenprogramm ist ein relativ kleines Programm (ca. 600 Zeilen in Pascal), das noch nichts Sinnvolles leistet, aber alle Macintosh-spezifischen Eigenschaften ausnutzt. Hierzu gehören z.B. die **Maus, PullDown-Menüs**, die **Fenster-Verwaltung** und die schnelle **Bildschirm-Grafik**. Der Name "Rahmenprogramm" kommt daher, weil dieses kleine Programm praktisch einen Rahmen (Skelet oder Gerüst) darstellt, der all das trägt, was das fertige Programm hinterher einmal leisten soll. Wenn dieser Teil des Programms erst einmal (korrekt) läuft, ist ein Großteil dessen, was ein Macintosh-Programm zum Macintosh-Programm macht, realisiert. Dieses Rahmenprogramm wird zunächst in einer sehr einfachen Version vorgestellt und im Rest des zweiten Teils dann nach und nach komplettiert.

Im Anhang werden dann noch eine Reihe von Hinweisen gegeben, wie die vorgestellten Beispiele in eine Reihe anderer Sprachen übertragen werden können und auf welche Probleme und Fallen man bei solchen Übertragungen besonders achten muß.

Welche Anforderungen dieses Buch an den Leser stellt:

Da dies — wie schon erwähnt — weder ein Lehrbuch der Programmierung noch ein Handbuch einer Programmiersprache ist, muß ich entsprechende Fähigkeiten beim Leser voraussetzen. Echten Nutzen bringen können die folgenden Beschreibungen allen, die mindestens geringe bis mittlere Erfahrungen mit der Programmierung "an sich" besitzen und auch schon kleinere Programme komplett selbst geschrieben haben.

Da alle Programmier-Beispiele in Pascal geschrieben sind, werden auch Grundkenntnisse dieser Programmiersprache nahezu überall benötigt. Ich habe mir bewußt Mühe gegeben, die Beispiele nicht zu sehr von Pascal abhängig zu machen und eine einfache Übertragbarkeit in andere Sprachen zu gewährleisten. An vielen Stellen sind Hinweise eingestreut, wie die entsprechenden Konstruktionen in anderen Programmiersprachen aussehen würden. Ein Anhang erklärt dann noch einmal ausführlich die Techniken, die bei einer Übertragung in andere Sprachen Verwendung finden müßten. Um eine solche Übertragung aber überhaupt sinnvoll versuchen zu können, müssen wenigstens rudimentäre Grundkenntnisse von Pascal vorhanden sein.

Und um die Beispiele wirklich ausprobieren zu können, muß natürlich auch eine Programmiersprache vorhanden sein. Dies dürfte aber wohl selbstverständlich sein. Den Umgang damit sollte man, am besten noch bevor man mit den ersten Beispielen aus diesem Buch beginnt, an kleinen Beispiel - programmen üben. Diese sollten noch keine Macintosh-spezifischen Besonderheiten enthalten, sondern am besten nur aus ein paar Zeilen bestehen, die auch auf jedem anderen Computer laufen würden.

Am Ende schließlich steht noch eine sehr große und schwer zu erfüllende Anforderung an den Leser und Programmierer: Geduld und die Bereitschaft zum Umdenken! Gerade wer schon einige Erfahrungen mit anderen Computern hat, wird eine ganze Weile benötigen, um die Denkweisen und Gewohnheiten, die er beim Programmieren üblicherweise anwendet, abzulegen.

Fast hätte ich jetzt noch eine Anforderung vergessen: Zähigkeit. Oberstes Motto bei der Programmierung ist immer: Bloß nicht aufgeben und immer eine Sicherungs-Kopie bereithalten. Gerade wenn man selbst zu experimentieren beginnt, gibt einem der Macintosh nicht nur eine Vielzahl neuer Möglichkeiten, sich auszudrücken, sondern auch eine Vielzahl neuer Möglichkeiten, Fehler zu machen! Die Bombe ("Schwerwiegender Systemfehler...") wird einem bald zum vertrauten Anblick werden.

Dortmund, im November 1985

markus breuer

1 Einleitung

Die Hardware, die mit dem Macintosh einem einzelnen Benutzer zur Verfügung steht, ist sicherlich erstaunlich — oder war es, als der Macintosh konzipiert und auf dem Markt eingeführt wurde. Für ca. 10.000,– DM bekommt man ein Computersystem, das einen der modernsten Micropro - zessoren auf dem Markt enthält und dessen Grafik, sowohl vom Tempo wie auch von der Raffinesse und den Möglichkeiten her, so manche fünfmal teurere Anlage schlägt. Natürlich gibt es inzwischen Systeme, die für noch weniger Geld noch leistungsfähigere Hardware bieten. Man sollte aber nie vergessen, daß, als der Macintosh eingeführt wurde, das billigste Computersystem mit ähnlicher Hardware immerhin gut das Doppelte kostete (die Lisa).

Was den Macintosh aber aus der Masse seinen Mitbewerber hervorhob und hervorhebt, ist nicht so sehr seine Hardware als vielmehr seine Software. Sämtliche (der wenigen zur Markteinführung erhältlichen) Programme waren leicht zu erlernen und auch später leicht zu bedienen. Erreicht wurde dies mit einer Reihe von Konzepten, die im folgenden noch vorgestellt werden, die aber alle auf einigen wesentlichen Grundprinzipien aufbauen:

- Abbildungen (Grafiken) statt schwerverständlicher Texte
- Eine einheitliche Metapher für die Bedienung des Computers
- Intuitiv erlernbare Handgriffe für die Bedienung
- Sicherheit vor Bedienungsfehlern des Benutzers
- "Freundliches, uncomputerhaftes" Aussehen
- weitgehende Standardisierung der Bedienung aller Programme

Diese Prinzipien allerdings sind nicht "in den Macintosh eingebaut". Jedes Programm, das neu geschrieben wird, muß explizit auf ihre Einhaltung achten. In den ROM (*Read Only Memory*, auf deutsch *Festwertspeicher*) des Macintosh wurden allerdings eine ganze Reihe von Hilfsprogrammen eingebaut, die die Programmierung solcher Programme vereinfachen sollten. Deshalb bleibt es für jeden Programmierer wichtig, sich einmal ganz ausführlich mit diesen Prinzipien, nach denen die Bedienung eines Programms (seine Benutzeroberfläche) gestaltet werden sollte, zu beschäf - tigen.

Sie sind in etwas expliziterer Form als in den sechs oben angeführten Punkten, in einem Kapitel in der Bibel der MacProgrammierer "*Inside Macintosh*" niedergelegt. In solcher Ausführlichkeit kann innerhalb dieses Buches allerdings nicht darauf eingegangen werden. In den folgenden Abschnitten der Einleitung soll allerdings in kurzer Form versucht werden, sie etwas näher zu erläutern.

Eine der besten Methoden, die Benutzeroberfläche eines eigenen Programms zu entwerfen, ist aber immer auch, sich ein gut geschriebenes Programm anzuschauen und ähnliche Bedienungselemente in seine eigenen Programme einzubauen. Nicht alle bereits erhältlichen Programme für den Macintosh können aber hierfür genommen werden. Nicht einmal die Programme von Apple selbst halten sich an alle Regeln. MacPaint z.B. verletzt viele davon und sollte nach Möglichkeit nicht als Vorbild herangezogen werden. Sehr gute Programme in dieser Hinsicht sind allerdings MacWrite und vor allem auch MacDraw.

Wenn ich übrigends bis jetzt davon gesprochen habe (und auch im folgenden noch öfter darauf kommen werde), daß eine bestimmte Eigenschaft des Macintosh vollkommen neuartig sei, so stimmt das meist natürlich nur innerhalb gewisser Grenzen. Schon vor dem Macintosh gab es die Lisa, und noch vor der Lisa gab es eine Reihe experimenteller Rechner vor allem der Firma XEROX, die die meisten der "neuartigen" Eigenschaften bereits in ähnlicher Form besaßen. Diese Rechner haben jedoch nie eine große Verbreitung oder einen hohen Bekanntheitsgrad erreicht. Deshalb scheint es mir berechtigt, auch die Fähigkeiten des Macintosh und seiner Software, die ihren Stammbaum mehr oder weniger direkt auf die entsprechenden XEROX-Systeme zurückführen können, als "neu" zu bezeichnen. Die großartigen Leistungen der Forscher am XEROX Palo Alto Research Center (PARC) sollen aber auf keinen Fall unerwähnt bleiben.

1.1 Die Bedeutung der Macintosh-Hardware

Die Hardware des Macintosh ist vollkommen auf die oben beschriebenen Prinzipien, die im wesentlichen in Software realisiert sind, abgestimmt. Dies unterscheidet Programme auf dem Macintosh auch nach wie vor von ähnlichen Programmen auf anderen Rechnern (z.B. MousePaint auf IBM PCs u.ä.), deren Hardware nicht so konsequent auf die Software abgestimmt ist. Beim Macintosh wurde die Software nicht für die Hardware geschrieben, sondern die Erfordernisse der Programme (die zunächst nur in den Köpfen der Designer existierten) bestimmten den Hardware-Entwurf.

Der Bildschirm des Macintosh spielt eine große Rolle bei all diesen Erwägungen. Der Bildschirm und die gesamte Video-Hardware sind konsequent für die Anwendung von hochauflösender *bitmapped* Grafik konzipiert. Während es auf anderen Rechnern einen Grafik-Modus (oder deren mehrere) und einen Text-Modus des Bildschirms gibt, gibt es auf dem Macintosh nur Grafik — auch Texte müssen auf den Bildschirm "gemalt" werden. Die Auflösung (Anzahl einzeln ansteuerbarer Punkte auf der horizontalen und vertikalen Achse des Bildschirms) ist dabei mit 512 mal 342 Punkten nicht einmal ungewöhnlich hoch. Trotzdem ist der Bildschirm des Macintosh im Vergleich mit anderen Rechnern, die "auf dem Papier" eine viel feinere Grafik haben, viel klarer und schärfer.

Um dem Benutzer ein von seinem normalen Schreibtisch vertrautes Bild geben zu können, wurde der Bildschirm von vorneherein auf eine Darstellung von schwarzen Abbildungen auf (blau-)weißem Hintergrund ausgelegt. Dies mit der erforderlichen Schärfe und Flimmerfreiheit zu ermöglichen, stellte besondere Anforderungen an die Video-Hardware. Jeder, der schon einmal solche Darstellungen (schwarz auf weiß) auf anderen Monitoren gesehen hat, die nicht speziell dafür ausgelegt sind, wird dies durch rasches Eintreten von Kopfschmerzen bestätigen können.

Um die hochauflösende Grafik auch für die Bedienung von Programmen einsetzbar zu machen und nicht nur für die Programm-Ausgaben, wurde es nötig, dem Benutzer ein Hilfsmittel an die Hand zu geben, mit dem er jeden einzelnen Punkt auf dem Bildschirm rasch und mühelos ansteuern kann. Die *Maus* erwies sich hier als eine nahezu ideale Lösung, da sie gleichzeitig schnelle und genaue Positionierung auf dem Bildschirm erlaubt (obwohl es für spezielle Anforderungen andere Hardware gibt, die in einzelnen Fällen einer Maus überlegen sind.

Um die hochauflösende Grafik auch schnell genug zu machen und die zur erwartenden großen und komplexen Programme unterstützen zu können, wurde eine CPU benötigt, die leistungsfähiger war als die bis dahin bei Microcomputern eingesetzten Microprozessoren. Daher wurde im Macintosh ein 16/32-Bit-Prozessor namens MC68000 als CPU eingesetzt.

Um auch gelegentliche Computerbenutzer anzusprechen und Leute, die nicht ihre Wohnungseinrichtung für den Computer umstellen wollen, mußte das gesamte Gerät relativ kompakt und leicht transportabel gehalten werden. Eine arbeitsfähige Grundkonfiguration des Macintosh (ohne Drucker) nimmt deshalb auf dem Schreibtisch kaum mehr Platz ein als zwei Blatt Papier und ist mit wenigen Handgriffen vom Schreibtisch ins Regal oder in den Kofferraum eines Autos gepackt.

Wenn man alle diese Punkte berücksichtigt, sieht man — meiner Meinung nach — daß die Hardware des Macintosh wirklich extrem konsequent auf die Bedürfnisse der Software bzw. des mit dieser Software arbeitenden Anwenders zugeschnitten ist. Obwohl inzwischen an anderen Stellen mehr Hardware für weniger Geld zu haben ist, fehlt diese letzte Konsequenz im Design vieler anderer Rechner.

1.2 Einsatz von Grafiken für die Bedienung eines Computers

Gerade auch der intensive Einsatz von Grafiken, Abbildungen, Bildern und Symbolen hat dem Macintosh am Anfang seiner Laufbahn den Ruf eingebracht, ein recht teures Spielzeug zu sein. Ernsthafte Computer (–Programme) hatten einfach nicht so auszusehen! Selbst wenn auf anderen Rechnern gelegentlich einmal die Grafikfähigkeiten genutzt wurden, so geschah dies meist innerhalb von Video-Spielen oder nur als Ausgabe-Möglichkeit für ernsthafte Business-Grafiken oder CAD-Systeme.

Grafiken und Symbole auch für die Bedienung von Programmen einzusetzen, war jedenfalls bis zur Einführung des Macintosh wenig gebräuchlich und ist von Anfang an mit viel Kritik bedacht worden. Bald stellte sich jedoch heraus, daß vieles, was auf anderen Computern durch Text-Ausgaben und textuelle Befehle erreicht wurde, durch grafische Lösungen viel besser zu erreichen war.

Wichtigster Gesichtspunkt bei der Verwendung von Grafik für die Bedienung von Programmen auf dem Macintosh war in erster Linie die leichtere Erlernbarkeit der Programme gewesen. Für viele Benutzer, die vorher nie mit anderen Computern in Berührung gekommen waren, ist es oft viel leichter, ein Ergebnis zu erzielen, indem sie eine bestimmte Bewegung mit der Maus machen oder mit ihr auf ein bestimmtes Symbol zeigen, als durch die Angabe eines auch noch so "mnemonisch" bezeichneten Befehls auf der Tastatur.

Diese Ausrichtung auf die einfache Erlernbarkeit der Bedienung hat manchmal negative Auswirkungen auf die spätere Bedienung des Programms. Viele Benutzer beschweren sich darüber, daß die einfachen Möglichkeiten Befehle zu erteilen ihnen zu langsam und umständlich sind, wenn sie ein Programm richtig beherrschen. Dies ist allerdings kein grundsätzlicher Mangel grafisch bedienter Programme. Prinzipiell können gut überlegte grafische Lösungen auch auf Dauer schneller und bequemer sein als ihre textuellen Alternativen. Die Tastatur sollte deshalb aber innerhalb keines Programms völlig vernachlässigt werden und jederzeit als zusätzliches,

wichtiges Hilfsmittel für die Bedienung eines Programms berücksichtigt werden.

Somit kommen wir zu ein paar wichtigen Grundregeln für den Einsatz der Grafik für die Bedienung von Programmen auf dem Macintosh (Der Begriff gibt, sollten diese genutzt werden.

• Wann immer es für die Darstellung eines bestimmten Programmzustands eine Möglichkeit für den Einsatz von Grafiken, Bildern und Symbolen gibt, sollten diese genutzt werden.

• Wann immer es, im Rahmen der sonstigen grafischen Gestaltung eines Programms, eine Möglichkeit gibt, bestimmte Bewegungen der Maus oder ein Deuten mit der Maus auf bestimmte Teile von Abbildungen, als Befehl des Anwenders an das Programm aufzufassen, sollte diese Möglichkeit berücksichtigt werden. Vorzugsweise sollten solche "grafischen Befehle" nicht ausschließlich, sondern alternativ zu anderen Möglichkeiten der Befehlserteilung genutzt werden.

• Befehle, die durch Mausbewegungen ausgelöst werden, sollten nicht nach der Befehlserteilung intensive Tastatureingaben verlangen. Wenn solche Befehle unbedingt noch weitere Angaben benötigen, sollten dem Anwender sinnvolle Voreinstellungen dafür bzw. die letzten Angaben wieder erneut vorgeschlagen werden, die mit einer weiteren Mausbewegung oder einem kurzen Tastendruck übernommen werden können.

• Wenn der Anwender aber in einem Moment, in dem er typischerweise einen Befehl benötigt, seine Hände nicht für die Maus frei hat (z.B. bei der Text-Verarbeitung), sollte es zur Befehlserteilung Möglichkeiten geben, die vorzugsweise die Tastatur nutzen; eventuell alternativ zu andern Möglichkeiten.

Wenn diese Regeln in einem Programm stets befolgt werden, so hat es gute Chancen, sowohl einfach zu erlernen zu sein, als auch hinterher den versierten Benutzer nicht zu bremsen. Im Zweifelsfall sollte aber immer grafischen, einfachen Lösungen der Vorzug vor einer vielleicht schnelleren, aber schwer zu erlernenden oder zu behaltenden Lösung gegeben werden.

Einige schöne Beispiele, wie der Einsatz von Grafik und Maus auch für einen erfahrenen Benutzer jeder Befehlserteilung über die Tastatur überlegen ist, hält der Finder bereit, die Betriebssystemoberfläche (auf computerdeutsch oft *Shell* genannt) des Macintosh. Selbst als jahrelanger Benutzer von mächtigen

Betriebssystemen auf Großcomputern ziehe ich die Macintosh-Variante eines Kopie-Befehls für Dateien jeder anderen Möglichkeit vor.

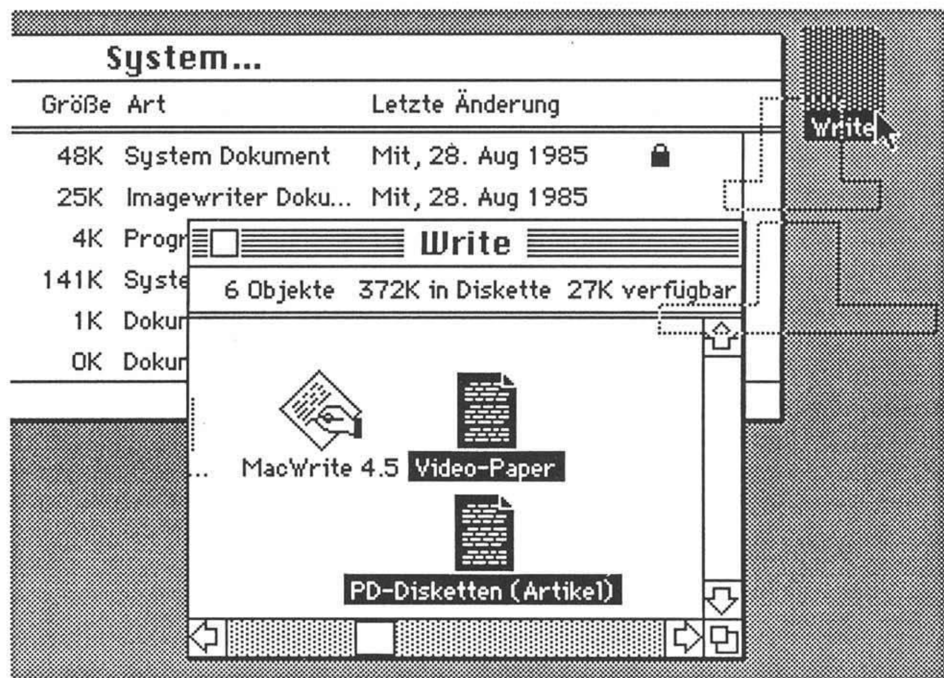


Bild 1 - 1: *Eine Gruppe von Dateien wird im Finder kopiert*

Der Anwender ergreift einfach die Symbole der Dateien mit der Maus und bewegt sie zu der anderen Diskette, auf die sie kopiert werden sollen. Auch das Löschen von Dateien, indem man sie in einen Mülleimer legt, hat etwas für sich. Man achte bei der obigen Abbildung auch noch auf ein anderes schönes Beispiel für den Einsatz von grafischen Symbolen. Im halb verdeckten Fenster "System..." wird eine tabellarische Auflistung von Dateien gegeben. Einige der Dateien sind vor einem versehentlichen Löschen geschützt. Der Finder symbolisiert dies, indem er bei deren Namen ein Vorhängeschloß zeigt.

1.3 Die einheitliche Metapher zur Bedienung eines Programms

Ein großer Vorteil der Bedienung des Macintosh gegenüber anderen Rechnern ist auch das einheitliche Modell (oft auch "Metapher" genannt), das dem Benutzer als Bild der Vorgänge im Rechner angeboten wird. Sobald der Benutzer dieses Modell erst einmal begriffen hat, kann er eine Reihe möglicher Aktionen (Befehle) und ihre Wirkung voraussehen, ohne ins Handbuch zu schauen.

Der Finder bietet dem Benutzer z.B. das Modell eines Schreibtisches an, auf dem eine Reihe Blätter Papier und Ordner liegen, in denen die Blätter abgelegt werden können. Die "Blätter" (Fenster) können sich überdecken und gegeneinander verschoben werden. Das Blatt, das jeweils von größtem Interesse ist, liegt an oberster Stell des imaginären Stapels und ist vollständig zu sehen. Jederzeit kann aber auch ein anderes Blatt an die oberste Stelle geholt werden.

Dieses Modell kann nicht vollständig auf die Bedienung jedes Programms übertragen werden. Viele typische "Handgriffe" bleiben aber gleich und werden auch in vielen anderen Programmen so oder ähnlich eingesetzt. Überlappende Fenster, die sich verhalten wie Blätter von Papier, werden in vielen Programmen verwendet. Die grundsätzliche Methode der Befehlsertei - lung ist ebenfalls in vielen Programmen ähnlich. Ein Objekt, bzw. dessen Abbild (oft ein sog. *Icon*) wird zunächst zur Bearbeitung ausgewählt, dann wird ihm ein Befehl erteilt. Die Befehlserteilung erfolgt über sogenannte "PullDown-Menüs", die erst dann vom oberen Rand des Bildschirms "herunterklappen", wenn sie benötigt werden.

Prinzipiell sollte jedes Programm versuchen, dem Benutzer ein einheitliches Modell der Vorgänge innerhalb des Datenbestandes anzubieten. Wenn irgendwie möglich, sollte dieses Modell grafisch gestaltet werden. Falls dieses Modell Tatbestände der Wirklichkeit auf dem Bildschirm zeigt, sollten sich diese Tatbestände und Objekte auch vergleichbar, wie ihre Gegenstücke in der Wirklichkeit, verhalten.

Hierzu ein Beispiel: Der Finder des Macintosh zeichnet eine modellhafte Schreibtisch-Umgebung auf dem Bildschirm. Zu dieser Umgebung gehört auch ein Papierkorb. Legt man eine Datei (bzw. deren Symbol) in den Papierkorb, so entspricht das dem Vorgang in der Wirklichkeit, ein Dokument fortzuwerfen. Überlegt es sich der Benutzer aber anders, so kann er das Dokument (die Datei) wieder aus dem Papierkorb hervorholen — sofern er nicht inzwischen geleert worden ist und sich das Dokument deshalb

überhaupt nicht mehr darin befindet. Den Inhalt des Bildschirm-Papierkorbs kann man sich jederzeit anschauen — wie den eines wirklichen Papierkorbs.

1.4 Intuitive, standardisierte Bedienung der Programme

Aus einem solchen einheitlichen Modell ergeben sich eine Reihe von Vorteilen für den Benutzer. Insbesondere wird das Erlernen eines Programmes zur intuitiven Angelegenheit, die mit wesentlich weniger Anleitung durch Handbücher oder andere Personen auskommt, als auf herkömmlichen Systemen. Der Benutzer wird angehalten, durch Experimentieren zu lernen.

Sobald er erst einmal ein Programm des Macintosh kennt, kennt der Benutzer viele der wichtigsten Eigenschaften der gesamten Benutzeroberfläche des Macintosh. Wenn er ein anderes Programm das erste Mal kennenlernt, kann er diese Kenntnisse sofort anwenden. Hierzu gehören wesentliche Grundkomponenten der Macintosh-Hard- und -Software wie die Maus, die Bildschirm-Grafik, die PullDown-Menüs und überlappende Bildschirm-Fenster etc. Bei den meisten Programmen kommen auch noch Ähnlichkeiten bei "höheren" Aspekten der Bedienung, wie der Befehlsstruktur, einzelnen Befehlsnamen oder der typischen Bedeutung eines Mausklicks, eines Doppel-Mausklicks usw. dazu.

Diese Ähnlichkeiten zwischen verschiedenen Programmen spielen auch dann eine große Rolle, wenn ein Benutzer häufig mit mehreren Programmen gleichzeitig arbeitet oder rasch zwischen ihnen wechselt. Wenn er dabei immer zwischen völlig verschiedenen Philosophien der Programm-Bedienung wechseln müßte, wäre das sicherlich eine große Belastung für ihn — und eine nicht zu unterschätzende Fehlerquelle.

Einen großen Anteil an dieser Standardisierung der Programme hat die ToolBox des Macintosh, eine Sammlung nützlicher Hilfs-Routinen für den Aufbau einer Benutzerschnittstelle, die im ROM jedes Macintosh jedem Programm immer zur Verfügung steht. "ToolBox" bedeutet in diesem Zusammenhang zu deutsch etwa die "Werkzeugkiste des Programmierers". Die ToolBox-Routinen helfen einem Programm beim Umgang mit den typischen Komponenten der Macintosh-Benutzerschnittstelle. Grafik, Mausbewegungen, PullDown-Menüs, Fenster und viele andere dieser Komponenten werden über ToolBox-Routinen kontrolliert. Diese erledigen dabei teilweise recht umfangreiche Aufgaben und verlangen für einen großen Teil des Standard-Verhaltens der betroffenen Komponenten nur ein geringes

Eingreifen von seiten des eigentlichen Programms. Dies hat Vorteile für den Programmierer, aber auch für den Anwender.

Einerseits müssen Programme, die die ToolBox benutzen, nicht jedesmal "das Rad neu erfinden", und andererseits verwenden sie natürlich "Bausteine", die jedes andere Programm auch verwendet, und ähneln diesen dadurch. Genau das waren auch die beiden wichtigsten Gesichtspunkte, unter denen der Macintosh-ROM entwickelt wurde. Kein Programmierer sollte sich deshalb scheuen, möglichst oft und reichlich Gebrauch von der ToolBox zu machen. Dieses Buch kann ihm dabei hoffentlich eine wichtige Hilfe sein.

• Wann immer ein Programmierer einen neuen Befehl (eine neue Abbildung etc.) in sein Programm einbaut, sollte er sich andere Programme daraufhin anschauen. Kennen andere Programme gleiche oder ähnliche Befehle, so sollten sich die Befehle des neuen Programms in ihrer Benennung und ihrer sonstigen Bedienung möglichst nahe an diese bereits bekannten Programme anlehnen. Gerade wenn diese(s) andere(n) Programm(e) weit verbreitet ist (sind) sollten vielleicht auch Aspekte übernommen werden, die man selbst für nicht so gut oder verbesserungswürdig hält, da sich ein Großteil der Benutzer bereits an sie gewöhnt haben wird.

• Eine Minimal-Anforderung an jedes Programm stellt die intensive Nutzung der ToolBox-Routinen dar, wann immer es sich anbietet. Hierdurch wird schon einmal ein Mindestmaß an Standardisierung unter den Programmen gewährleistet. Durch die Nutzung der ToolBox kommen jedem Programm auch sofort sämtliche späteren Verbesserungen (oder Anpassungen des Macintosh an andere Muttersprachen seiner Benutzer) zugute — ohne daß am eigentlichen Programm auch nur kleine Änderungen durchgeführt würden.

1.5 Keine unterschiedlichen Programm-Zustände!

Mit die wichtigste Eigenschaft der meisten Macintosh-Programme ist, daß sie weitgehend ohne *Modi* auskommen. Ein *Modus* ist in diesem Zusammenhang ein bestimmter Zustand des Programms, in dem die Eingaben und Befehle des Benutzers anders interpretiert werden als in einem anderen Modus. Der Benutzer muß sich also stets des aktuellen Modus bewußt sein, in dem sich das Programm befindet, sonst wird er die Wirkung seiner Aktionen nie vorher einschätzen können. Nicht alle Macintosh-Programme vermeiden solche Programm-Zustände völlig, aber sie vermeiden sie meist doch in viel höherem Maße als vergleichbare Programme auf anderen Rechnern oder verbergen sie vor dem Benutzer und mildern ihren Einfluß auf das Verhalten des Programms stark ab.

Diese Eigenschaft, ohne Modi auszukommen, ist natürlich zum großen Teil durch Design-Entscheidungen in einem sehr frühen Stadium der Programmentwicklung bestimmt. Dazu müßte eigentlich einiges mehr gesagt werden, als an dieser Stelle möglich ist. Hier wird die Umstellung manchem, der bereits Erfahrungen mit der Programmierung anderer Rechner hat, recht schwer fallen. Ein großes Stück auf dem Weg zu einem Programm ohne Zustände ist jedoch bereits gegangen, wenn das Standard-Rahmenprogramm verwendet wird, wie es im zweiten Teil dieses Buches vorgestellt wird. Es handelt sich dabei im wesentlichen um ein Hauptprogramm und ein paar Hilfsroutinen, die zusammen dem Programmierer einen großen Teil der Arbeit mit dem Macintosh abnehmen.

Die Verwendung dieses Rahmenprogramms schränkt die Möglichkeiten des Programmierers kaum ein — nur das Programmieren von Zuständen wird wesentlich erschwert.

Das Erstaunliche für die meisten Programmierer ist, daß sich Zustände bald als wirklich entbehrlich erweisen, wenn man eine gewisse Zeit Übung auf dem Macintosh hat. Während man sich am Anfang noch dazu zwingen muß, ohne sie auszukommen, ergibt es sich später ganz natürlich. Wieder ganz wichtig ist dabei die Verwendung des Rahmenprogramms. Wer sich dieses genauer anschaut und vor allem zu verstehen versucht, wieso es so und nicht anders programmiert sein muß, wird auf Programm-Zustände bald verzichten können.

• Wann immer ein Programmierer oder Programm-Designer meint, einen Zustand in ein Programm einführen zu müssen, sollte er sich diese Sache mehr als zweimal überlegen. Zustände tragen wesentlich zur Schwierigkeit und Gefährlichkeit eines Programmes bei.

Zustände können bei Verwendung des später noch vorgestellten Rahmenprogramms sehr leicht daran erkannt werden, daß entweder das Hauptprogramm längere Zeit umgangen wird oder Variablen eingeführt werden, in denen Informationen über den Zustand des Programms gespeichert werden. Solche Variablen sind nur solange "O.K.", wie von ihrem Inhalt nicht das Verhalten des Programms nach außen — gegenüber dem Benutzer — abhängig gemacht wird.

• Die einzige Stelle, an der wirklich so etwas wie ein Wechsel des Programm-Zustands stattfinden darf, ist der Wechsel von einem Fenster zum andern. Aber auch in diesem Fall sollten nur die Aspekte des Programms, die vom Inhalt des aktuellen Fensters wirklich abhängig sind, auf diesen Wechsel

reagieren dürfen. Diese Programmteile, die *fensterabhängig* sind, sollten möglichst klein gehaltén werden.

1.6 Sicherheit der Bedienung

Sehr wichtig bei benutzerfreundlichen Programmen ist immer auch, daß sie den Anwender weitgehend vor seinen eigenen Fehlern schützen, ohne ihn aber zu gängeln! Dies kann entweder dadurch geschehen, daß Fehler vermieden werden oder die Folgen von Fehlern begrenzt werden und leicht wieder rückgängig zu machen sind.

Eine wichtige Eigenart der meisten Macintosh-Programme, durch die Fehler vermieden werden, ist bereits, daß sie ohne Modi auskommen. Im allgemeinen kann der Benutzer nie deshalb etwas falsch machen, weil er meint, in einem anderen Programmzustand zu sein, wie er wirklich ist.

Zudem ist die gesamte Art der Bedienung des Macintosh bereits weniger fehleranfällig als die mehr Kommando-orientierte Bedienungsweise anderer Rechner. Die Menüs zeigen zu jedem Zeitpunkt alle möglichen Befehle an. Der Benutzer kann deshalb gar keinen Befehl aufrufen, der zu einem gewissen Zeitpunkt nicht zulässig ist. Dieser Schutz kann noch dadurch verstärkt werden, daß man einzelne Punkte der PullDown-Menüs oder ganze Menüs nicht wählbar macht. Sie erscheinen dann grau und zeigen so dem Benutzer, daß im Moment dieser Befehl sinnlos oder unmöglich ist.

• Wenn ein Menü-Befehl also zu einem bestimmten Zeitpunkt nicht sinnvoll wählbar ist, sollte das Programm dies dem Benutzer zumindest dadurch anzeigen, indem es den entsprechenden Befehl (mit Hilfe der ToolBox) grau erscheinen läßt. Eine vielleicht noch bessere Lösung — die allerdings auch etwas mehr Programmaufwand verlangt — ist es aber in manchen Fällen, den Benutzer den Befehl zunächst wählen zu lassen, ihm aber dann zu sagen, warum der Befehl im Moment nicht sinnvoll ist. Auf keinen Fall sollte man den Benutzer aber einen Befehl wählen lassen, worauf einfach nichts geschieht.

Eine weitere Möglichkeit, Benutzer-Fehler abzufangen, sind die Dialoge des Macintosh; formularähnliche Gebilde, die in eigenen Fenstern auftauchen, sobald kompliziertere Anfragen an den Benutzer gerichtet werden, und wieder verschwinden, wenn sie nicht mehr benötigt werden. Solche Dialog-Formulare können mit verschiedenen Mitteln der ToolBox so gestaltet werden, daß auch bei komplexeren Eingaben Fehler nahezu unmöglich sind. So wird z.B. die Auswahl einer bestimmten unter einer Vielzahl von Wahl -

möglichkeiten in Dialogen typischerweise durch Ankreuzen der gewünschten Möglichkeit erreicht — eine viel weniger fehleranfällige Methode, als den Namen der Wahlmöglichkeit über die Tastatur einzugeben.

Bild 1 - 2: *Ein einfaches Dialog-Formular*

Dialoge bieten noch eine ganze Reihe anderer Möglichkeiten, den Benutzer vor seinen eigenen Fehlern zu bewahren und ihm vor allem auch sofort zu melden, was genau an seiner letzten Aktion falsch war.

• Wann immer ein Programm größere Mengen von Angaben vom Benutzer erfragen muß, sollten hierfür Dialoge eingesetzt werden. Um die Bearbeitung dieser Formulare nicht unnötig langwierig zu gestalten, sollten alle Eingabefelder und Ankreuz-Kästchen eines Dialogs sofort nach seinem Erscheinen mit möglichst "plausiblen" Voreinstellungen oder mit den zuletzt eingegebenen Werten gefüllt sein.

Eine ganz wichtige Eigenschaft der Macintosh-Benutzerschnittstelle ist auch der ***Undo***-Befehl, der in vielen Programmen zu finden ist. Immer wenn größere Änderungen am Datenbestand eines Programms durchgeführt wurden und dem Benutzer plötzlich einfällt, daß er sich beim letzten Befehl geirrt hat, ist der letzte Zustand vor der Änderung nicht verloren. Indem der Benutzer den Befehl ***Undo*** (auf deutsch ***Widerrufen***) aus den Menüs auswählt, wird die Wirkung des letzten Befehls rückgängig gemacht. Entscheidet er sich nun, daß der Einsatz dieses Befehls doch sinnvoll war, so kann er mit dem Befehl ***Redo*** (auf deutsch ***Wiederholen***) die Wirkung des widerrufenen Befehls sofort wieder eintreten lassen.

Ganz davon abgesehen, daß diese Möglichkeit unschätzbare Vorteile im täglichen Umgang mit einem Programm hat, eröffnet sie völlig neue Möglichkeiten beim Erlernen eines Programms. Der Benutzer kann einfach "drauflos experimentieren", ohne sich um die Folgen seines Tuns Gedanken machen zu müssen. Wann immer er merkt, daß er einen falschen Weg gegangen ist, kann er stets zum letzten Punkt zurückkehren. Eine gut realisierte ***Undo***-Funktion trägt wesentlich zur komfortablen Bedienung eines Programms bei. Manche Probleme eines Benutzers beim Umgang mit einem Programm, z.B. mit einer schwer verständlichen Befehlsstruktur, wiegen nur halb so schwer, wenn immer eine komplette ***Undo***-Möglichkeit besteht.

• Prinzipiell sollte jeder Befehl eines Programms, der schwerwiegende Änderungen am Datenbestand des Programms verursacht, widerrufbar sein. Als schwerwiegend möge hierbei jeder Befehl gelten, der nicht mit einem oder zwei anderen Befehlen oder Tastendrücken sowieso problemlos rückgängig zu machen ist. Jeder Befehl, der widerrufbar ist, sollte die Möglichkeit bieten, ihn einfach zu wiederholen (***Redo***), nachdem er rückgängig gemacht wurde.

• Wenn ein Befehl, der schwerwiegende Änderungen am Datenbestand des Programms verursacht, nicht widerrufbar ist, sollte der Benutzer vor Ausführung des Befehls gewarnt werden. Zusammen mit dieser Warnung sollte dem Benutzer die Gelegenheit gegeben werden, die Ausführung des Befehls abzubrechen.

1.7 Schlußbemerkung

An dieser Stelle konnten unmöglich alle Gesichtspunkte abgehandelt werden, unter denen die Benutzer-Oberfläche eines Macintosh-Programms gestaltet werden sollte. Ich hoffe aber, einen kleinen Überblick darüber gegeben zu haben, welches die wichtigsten Besonderheiten von Macintosh-Programmen sind. Es ist nicht allein die oft sehr grafisch orientierte Bedienung, die diese Programme zu guten Macintosh-Programmen macht. Es müssen eine Vielzahl anderer Punkte berücksichtigt werden, um z.B. ein Programm zu entwickeln, das so leicht zu erlernen und zu bedienen ist wie MacWrite.

Die Phase des Entwurfs einer Benutzer-Schnittstelle für ein neues Programm ist mit eine der wichtigsten bei der Programmentwicklung. Entscheidungen, die hier — bevor die erste Zeile des Programms geschrieben wird — getroffen werden, haben oft entscheidenden Einfluß über die Qualität und Brauchbarkeit des fertigen Programms.

Andererseits müssen die hier angeschnittenen Prinzipien auch bei der späteren Programm-Entwicklung immer wieder berücksichtigt werden. Bei den vielen kleinen Entscheidungen, die anstehen, wann immer ein neues Modul geschrieben wird oder ein schon geschriebenes geändert wird, sollten immer auch die Grundsätze für den Entwurf der Benutzerschnittstelle eine große Rolle spielen. Insbesondere die Einheitlichkeit der Bedienung innerhalb des neuen Programms selbst und auch im Vergleich zu anderen Programmen muß in diesen Fällen immer angestrebt werden.

2 Speicherverwaltung

Speicherverwaltung (auf neudeutsch "Memory-Management") ist für fast jedes Programm auf jedem Computer wichtig. Ganz besonders wichtig wird sie allerdings dann, wenn es eng wird im Speicher. Auf dem Macintosh (besonders mit nur 128K) ist der Platz erstens recht knapp, und zweitens ist es immer von Vorteil, die Eigenheiten (Vor- und Nachteile) der Speicher - verwaltung auf einem Rechner zu kennen, um ihn optimal zu nutzen. Drittens sind diese Eigenheiten gerade auf dem Macintosh mit Gefahren und Problemen verbunden, die aber nur dann auftauchen, wenn man sie nicht kennt oder nicht richtig verstanden hat.

Deshalb möchte ich — bevor ich näher auf die speziellen Funktionen des Macintosh MemoryManagers eingehe — etwas weiter ausholen und die verschiedenen Arten, wie Daten im Speicher abgelegt werden können, etwas näher betrachten. Übrigens liegen ja nicht nur die Daten, sondern auch die Programme, die mit diesen Daten arbeiten, im Speicher. Und um die Situation zusätzlich zu verkomplizieren, betrachten Teile der ToolBox auch Programmstücke als Daten.

Wir wollen uns hier aber nur mit der Sicht des Speichers als Behälter für Daten beschäftigen. Insbesondere soll es um die verschiedenen Arten gehen, wie Daten im Speicher abgelegt werden und verwaltet werden können. Auf dem Macintosh gibt es insgesamt 4 Daten-Sorten:

- globale Variablen
- lokale Variablen
- dynamisch angelegte festliegende Speicherblöcke
- dynamisch angelegte verschiebbare Speicherblöcke

Sobald einmal die Unterschiede zwischen diesen 4 Sorten klar sind, können die Vor- und Nachteile der einzelnen Speicherungsarten abgewägt werden.

2.1 Das Variablenkonzept

Variablen kennt an sich jede höhere Programmiersprache, und selbst Assembler-Sprachen kennen ähnliche Konstruktionen, um Speicherstellen mit sinnvollen Namen zu versehen. Wie Variablen gehandhabt werden können, hängt stark von der verwendeten Programmiersprache ab, deshalb will ich mich hier nicht zu ausführlich darüber auslassen. In den meisten Programmiersprachen bedeuten Variablen aber nicht nur einen Namen für eine bestimmte Speicherstelle, sondern der Compiler bzw. Interpreter verbindet gleichzeitig einen *Typ* mit dieser Variablen und achtet darauf, daß ihr nur Werte dieses Typs zugewiesen werden können. Zahlen können nicht Text-Variablen zugewiesen werden und Texte nicht numerischen Variablen, selbst wenn dieser Text z.B. "12345" lautet.

```
Mitgliederzahl

[25469]          <--- ganze Zahl

Nachname

[ M | e | i | e | r ] <--- Text
```

Bild 2 - 1: *Variablen*

Außer dieser Unterscheidungsmöglichkeit von Variablen (dem Typ) gibt es noch eine ganze Reihe weiterer möglicher Unterschiede zwischen Variablenformen. Es gibt z.B. in vielen Programmiersprachen Unterschiede zwischen *globalen* und *lokalen* Variablen. Generell sind es zwei Merkmale, die Variablen charakterisieren: Ihre Sichtbarkeit und ihre Lebensdauer.

Einige Arten von Variablen sind z.B. nicht im ganzen Programm sichtbar, sondern nur in Teilen davon (z.B. Prozeduren und Modulen), d.h. sie können nur in bestimmten Teilen eines Programmes verwendet werden und in anderen nicht.

Ein anderes Unterscheidungsmerkmal ist die Lebensdauer. Manche Variablen "leben" ein ganzes Programm lang, d.h. während des ganzen Programmlaufs ist eine Stelle im Speicher für den Wert dieser Variablen reserviert. Andere Variablen existieren nur, solange ein bestimmtes Unterprogramm läuft. Ist dieses beendet, so wird der Speicherplatz, den sie eingenommen haben, nicht mehr reserviert und kann für andere Zwecke verwendet werden.

Sichtbarkeit und Lebensdauer sind bei vielen Variablen oft gleich. Solange sie sichtbar sind, ist auch Speicher für sie reserviert, und sobald sie nicht mehr verwendet werden können (weil sie unsichtbar geworden sind), kann der Speicherplatz, den sie eingenommen haben, für andere Zwecke verwendet werden. Bei anderen Variablenarten sind Sichtbarkeit und Lebensdauer getrennt. Dies sind die komplizierteren Fälle, die Sorgenkinder eines Programms, auf die jeder Programmierer verstärkt achtgeben muß.

2.2 Globale Variablen

Globale Variablen sind wohl jedem Benutzer jeder Programmiersprache bekannt. Sie sind die einfachste Realisierung des Variablenkonzeptes. Einfache "höhere" Programmiersprachen wie die meisten Varianten von BASIC kennen nur globale Variablen. Was sie auszeichnet, ist, daß sie im ganzen Programm sichtbar sind und während des gesamten Programmlaufs existieren. Globale Variablen müssen im allgemeinen eindeutige Namen haben, d.h. es kann keine zweite globale Variable gleichen Namens in einem Programm geben. Bei compilierten Sprachen legt der Compiler die physikalische Adresse, die diesem Namen entspricht, meist bei der Übersetzung des Programms fest, und sie ändert sich während des ganzen Programmlaufs nicht. Bei dieser Adresse ist dann ein Bereich von festgelegter Größe für den Wert dieser Variablen reserviert.

Beim Macintosh liegt der Fall etwas komplizierter. Damit Programme nicht darauf angewiesen sind, einen bestimmten Speicherbereich frei vorzufinden, wenn sie starten, werden die Programme üblicherweise so geschrieben, daß die Adressen der globalen Variablen erst beim Start des Programms bestimmt werden. Die Namen der globalen Variablen stehen nicht mehr für eine bestimmte Speicheradresse, sondern für den Abstand (engl. *Offset*) dieser Variablen vom Beginn der globalen Variablen. Beim Start reserviert sich jedes Programm nun soviel Platz, wie es benötigt, und merkt sich den Beginn dieses reservierten Blocks. Wird während des Programmlaufs die Adresse einer Variablen benötigt, addiert das Programm dessen *Offset* auf die Startadresse der globalen Variablen auf und erhält so die wirkliche Adresse.

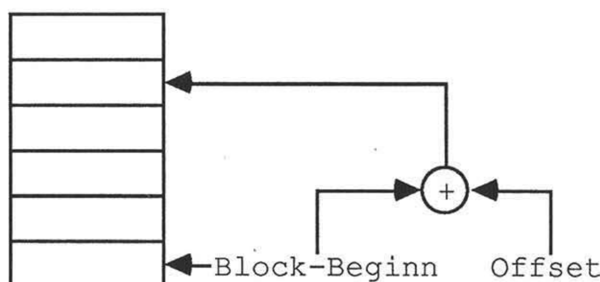


Bild 2 - 2: *Berechnung der Adresse einer Variablen*

Die Basisadresse der globalen Variablen liegt dabei übrigens auf dem Macintosh immer im Register A5 der CPU. Durch die Nutzung des Adressierungsmodus *Adreßregister plus Offset* des MC68000 kann die oben beschriebene Addition mit sehr geringen Zeitverlusten erfolgen. Ein Vorteil dieses Prinzips, der allerdings im Moment noch kaum genutzt wird, ist es, daß die Daten eines Programmes an nahezu beliebiger Stelle im Speicher stehen können und sogar während des Programmablaufs verschoben werden können. Es können auf diese Weise auch gleichzeitig mehrere Programme, jedes mit seinem eigenen Satz globaler Variablen, im Speicher sein. Bei jedem Wechsel von einem Programm zum anderen wird dann der Inhalt des Registers A5 entsprechend getauscht.

2.3 Lokale Variablen

Globale Variablen bringen eine ganze Reihe von Problemen mit sich. Gut strukturierte Programme werden ja — sofern es die Programmiersprache zuläßt — in eine ganze Reihe von Teilen (z.B. *Unterprogramme*, *Prozeduren* oder *Funktionen* genannt) zerlegt. Innerhalb dieser Prozeduren werden natürlich auch Variablen verwendet. Viele Variablen werden sogar nur während des Ablaufs einer Prozedur benötigt und davor und danach nie wieder. Es wäre nun eine horrende Verschwendung von Speicherplatz, wenn der Compiler oder Interpreter den Platz für diese Variablen während des ganzen Programmlaufs reservieren würde, obwohl er doch nur kurze Zeit benötigt wird.

Zusätzlich kommt noch dazu, daß es ein Programm nicht gerade übersichtlich macht, wenn alle Variablen, die es benötigt, global sind. Prozeduren werden

oft viel besser verständlich, wenn die Variablen, die sie benötigen, auch direkt bei ihnen definiert (und kommentiert) werden. So ist es z.B. auch möglich, in zwei Prozeduren Variablen gleichen Namens zu verwenden. Hat man nur globale Variablen zu Verfügung, muß man sich für jede neue Variable einen neuen Namen "aus den Fingern saugen". Dies führt in Verbindung mit der Beschränkung der meisten Compiler und Interpreter auf eine maximale signifikante Namenslänge meist zu recht kryptischen Abkürzungen.

Zur Lösung all dieser Probleme hat man das Konzept der lokalen Variablen entwickelt. Lokale Variablen "leben" nur während der Laufzeit des Programmstückchens, in dem sie definiert sind. Sie sind auch nur in diesem Stückchen sichtbar. In einigen Sprachen ist das ein Unterprogramm, in der Programmiersprache C z.B. können es auch noch kleinere Stückchen, wie etwa Programm-Schleifen, sein. In Pascal sind alle Variablen lokal, die nicht im Hauptprogramm deklariert werden.

Immer wenn eine Gruppe von lokalen Variablen benötigt wird (es sind meistens mehrere auf einmal), wird ein Speicherblock dafür reserviert, dessen Größe natürlich abhängig vom Typ und der Anzahl der neu benötigten Variablen ist. Der Beginn dieses Speicherblocks wird in einer besonderen Variablen gemerkt. Die Namen von lokalen Variablen bedeuten jetzt den Abstand, den die entsprechende Variable vom Beginn dieses Blockes hat. (Ähnlich wie die Namen globaler Variablen einen Abstand vom Beginn des Speicherblocks für globale Variablen bedeuten.) Die vollständige Adresse der (lokalen) Variablen wird erst bei Bedarf berechnet, indem dieser Abstand auf den Block-Beginn aufaddiert wird. Sobald die Prozedur, in der die lokalen Variablen definiert wurden, beendet ist, wird der Speicherblock wieder freigegeben und kann von der nächsten Prozedur wiederverwendet werden.

Als einfachste Möglichkeit, das Reservieren und Freigeben zu realisieren, hat sich ein sogenannter *Stapel* (engl. *Stack*) erwiesen. Hierbei bedient man sich zweier spezieller Variablen, die auf den Anfang und das Ende des gerade gültigen Speicherblocks für lokale Variablen zeigen. In der Variablen SP (*StackPointer* bzw. Stapelzeiger) merkt man sich das Ende dieses Blocks und in FP (*FramePointer* = Zeiger auf den Beginn der lokalen Variablen) den Anfang des Blockes. SP ist gleichzeitig das Ende des bisher vom Programm benötigten Speichers. Der Platz darüber ist noch frei.

Wird ein neuer Block von **n** Bytes Größe für lokale Variablen benötigt, so wird einfach der FP auf die Speicherstelle eins hinter den aktuellen SP gesetzt und dort der alte Wert von FP hinterlegt. SP wird dann um n+1 erhöht. Die

über FP liegenden Speicherstellen stehen jetzt für die lokalen Variablen zur Verfügung.

Werden die lokalen Variablen nicht mehr benötigt, so wird einfach der SP auf die Speicherstelle eins unter den aktuellen FP gesetzt und der FP wieder mit dem alten Wert geladen. Der Speicherblock, in dem gerade noch die lokalen Variablen lagen, steht jetzt wieder für andere Zwecke zur Verfügung.

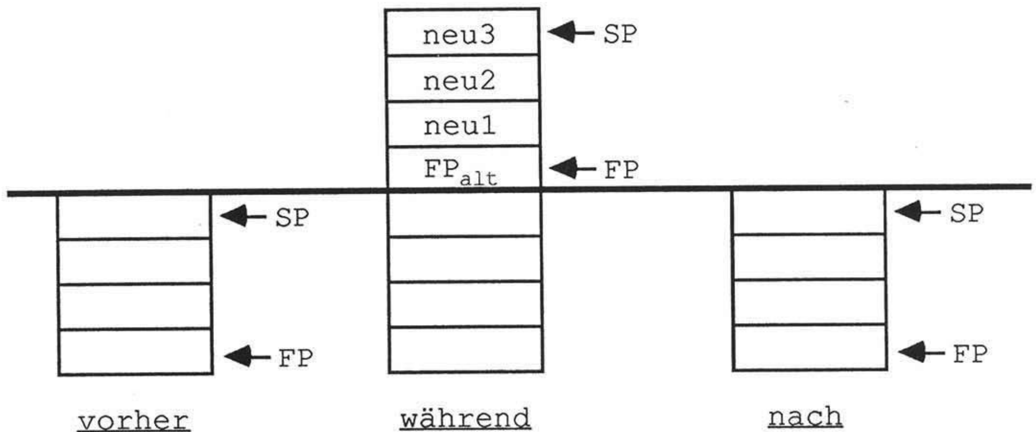


Bild 2 - 3: *Der Laufzeit-Stack*

Auf diese Art und Weise erreicht man, daß der Platz für die Variablen nur so lange reserviert bleibt, wie sie auch gebraucht werden. Variablen, die nie zur selben Zeit "leben", können sich so gewisse Speicherstellen "teilen"; zu verschiedenen Zeitpunkten stehen ganz verschiedene Variablen an ein und derselben Stelle.

Die hier beschriebene Art, lokale Variablen zu verwalten, ist ziemlich Computer- und Sprach-unabhängig und wird auf vielen Computern so oder so ähnlich angewendet. Leichte Abweichungen davon stellen oft Zugeständnisse an eine oft ungenügende Hardwareunterstützung für diese Konstruktion dar. Auf dem Macintosh wird sie jedoch von den meisten Programmen "in Reinkultur", so wie hier beschrieben, angewendet.

Als FramePointer findet dabei üblicherweise das Prozessorregister A6 Verwendung, und der StackPointer wird im Register A7 (das auch oft SP genannt wird) abgelegt. Die flexiblen Adreß-Modi des MC68000 ermöglichen auf diesem Wege eine sehr effiziente Realisierung von lokalen Variablen. Beim MC68000 wächst der Stack übrigens von "oben nach unten", d.h. wenn neue Daten daraufgelegt werden, wird sein Inhalt (eine Speicher-Adresse) kleiner. Bild 2 - 3 muß man sich also eigentlich auf den Kopf gestellt betrachten.

2.4 Dynamisch angelegte festliegende Datenblöcke

Lokale Variablen werden in gewisser Weise dynamisch erzeugt. Sie existieren noch nicht zu Beginn eines Programmlaufs, sondern erst dann, wenn das Programm in den Programmteil, in dem sie definiert sind, eintritt. Sie verschwinden nach dem Austritt aus diesem Teil wieder und beanspruchen keinen Speicherplatz mehr. Wie wir sehen, liegt hier ein direkter Zusammenhang zwischen Sichtbarkeit und Lebensdauer vor.

Oft ist es jedoch nötig, oder zumindest praktisch, frei über die "Lebensdauern" von Variablen verfügen zu können. Man benötigt manchmal Datenstrukturen, die unabhängig vom Eintritt in einen bestimmten Programmteil erzeugt werden und deren Lebensdauer nicht mit der Dauer einer Prozedur zusammenhängt — aber deshalb trotzdem nicht gleich das ganze Programm überdauern soll.

Um dies zu erreichen, reserviert man üblicherweise beim Start des Programms einen mehr oder weniger großen Block im Speicher für diese dynamisch angelegten Daten. Dieser Block trägt meist den Namen *Heap* (engl. für "Haufen"). Benötigt man nun während des Programmlaufs einen Datenblock einer bestimmten Größe, so teilt man ein Stück des Heaps dafür ab und merkt sich die Adresse dieses Stücks in einer bestimmten Sorte von Variablen: einer *Zeiger-Variablen.* Zeiger-Variablen enthalten nicht selbst die Daten, die das Programm interessieren; sie *zeigen* nur auf das Stück im Heap, wo diese liegen. In Pascal verdeutlicht man dies dadurch, daß man den Wert dieser Daten nicht mehr dadurch im Programm ansprechen kann, indem man einfach ihren Namen schreibt, sondern hinter diesen Namen noch ein Zeigersymbol '^' setzen muß.

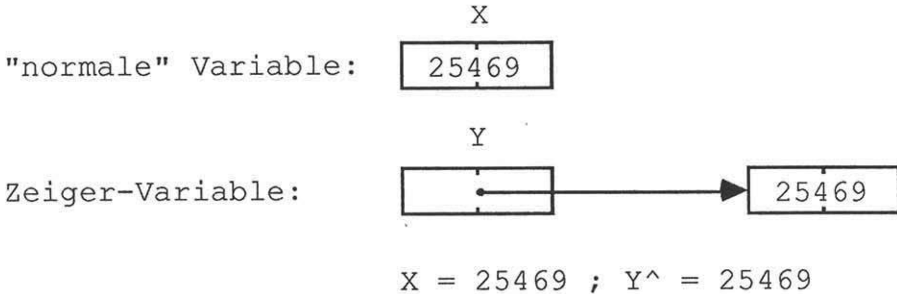


Bild 2 - 4: *"Normale" und Zeiger-Variablen*

Viele Programmiersprachen oder Betriebssysteme nehmen einem die Heapverwaltung glücklicherweise ab. "Glücklicherweise" deshalb, da es gar nicht so einfach ist, einen Heap effizient zu verwalten, wie dies vielleicht jetzt den Anschein hat.

In Pascal schreibt man z.B. einfach **New(zeigerVariable),** um einen Speicherblock der passenden Größe zu reservieren und gleichzeitig einen Zeiger darauf in **zeigerVariable** abzulegen. Benötigt man die Daten "am Ende des Zeigers" dann nicht mehr, lautet der entsprechende Aufruf **Dispose(zeigerVariable)**. Danach wird der von dieser Variablen belegte Platz freigegeben und steht eventuell wieder für die Anforderung von neuen Blöcken mit **New** zur Verfügung. Genau von diesem Vorgang des relativ ungeordneten und unvorhersehbaren Anforderns und Wiederfreigebens von Speicherblöcken hat der Heap (Haufen) seinen Namen. Auf den Stapel kann man nur oben etwas drauflegen oder herunternehmen. Im Haufen kann man so lange wühlen, bis man einen freien Platz gefunden hat, egal wo er liegt. Während beim Stapel der Zeiger SP sauber den bereits in Gebrauch befindlichen und noch freien Speicher trennte, liegen im Heap benutzte und unbenutzte Blöcke von Speicherzellen meist wild durcheinander.

Mit der Zeit ergeben sich deshalb bei wiederholten **New-** und **Dispose**-Aufrufen fast zwangsläufig neue Probleme. Wenn die mit **New** angelegten Blöcke nämlich nicht alle dieselbe Größe haben, paßt nicht in jeden freien Block jeder neu angefordete Block hinein. Nehmen wir einmal an, wir hätten zunächst mit **New** den ganzen Speicher gefüllt und dann 3 Speicherplätze freigegeben. Wir brauchen jetzt aber einen neuen Block der Größe 3. Obwohl wir genau wissen, daß 3 Plätze frei sein müssen, braucht der entsprechende Aufruf doch nicht unbedingt zu klappen. Bild 2 - 5 zeigt, warum.

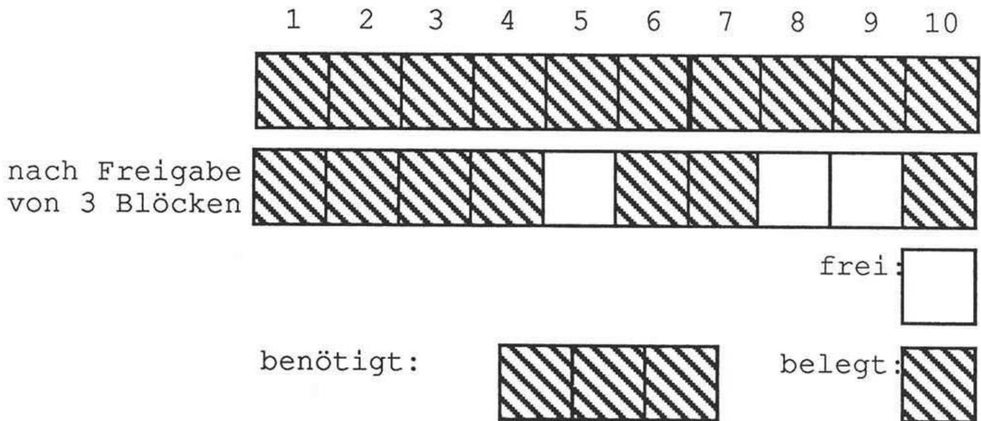


Bild 2 - 5: *Ein Problem im Heap*

Wir haben zwar 3 Plätze frei, diese liegen aber nicht zusammenhängend. Was wir brauchen, ist also ein zusammenhängender Block von mindestens der Größe 3.

Die stapelartige Verwaltung der lokalen Variablen garantiert uns, daß freie Speicherplätze immer beisammen liegen und freigegebene Plätze von jeder neuen Variablen wieder verwendet werden können. Der Heap kann dies nicht garantieren. Im Extremfall kann der halbe Heap frei sein und doch kein zusammenhängender Block von nur 2 Speicherstellen zur Verfügung stehen. Diesen Vorgang nennt man im allgemeinen "Fragmentierung", weil dabei der Heap in immer kleinere Stückchen (Fragmente) zerfällt, die nutzlos sind. Wir erkaufen uns also die erhöhte Flexibilität von dynamisch im Heap angelegten Variablen gegenüber den lokalen Variablen mit einer im Extremfall verschwenderisch schlechten Speicherausnutzung.

Zum Schluß dieser Betrachtung der Zeiger-Variablen muß aber auch noch erwähnt werden, daß Zeiger nicht zwangsläufig auf Variablen im Heap verweisen müssen. In fast allen Computer-Sprachen ist es auch möglich, Zeiger auf andere lokale oder globale Variable verweisen zu lassen. Die Hauptbedeutung von Zeigern liegt aber zweifellos in ihrer Verwendung zusammen mit dynamisch angelegten und zerstörten Variablen im Heap.

2.5 Dynamisch angelegte verschiebbare Datenblöcke

Auf dem Macintosh gibt es noch eine ganz besondere Art von dynamisch angelegten und zerstörten Datenblöcken, die auf anderen Rechnern kaum anzutreffen sind: die sogenannten "Handles" (zu deutsch: Griffe). Wie wir ja im Abschnitt über Zeigervariablen gesehen haben, kann die intensive Verwendung von dynamisch angeforderten Datenblöcken mit verschiedenen Größen dazu führen, daß neue Speicher-Anforderungen an den Heap nicht mehr befriedigt werden können, obwohl noch genug Platz vorhanden wäre — nur eben in kleinen Stücken. Könnte man diese kleinen Stücke zusammenfassen, wäre das Problem aus der Welt. Bild 2 - 6 zeigt, wie eine solche Kompaktierung beim Beispiel aus Bild 2 - 5 aussehen könnte. Alle belegten Blöcke werden am unteren Ende des Heaps zusammengefaßt, und am oberen Ende ergibt sich ein genügend großer, freier Bereich.

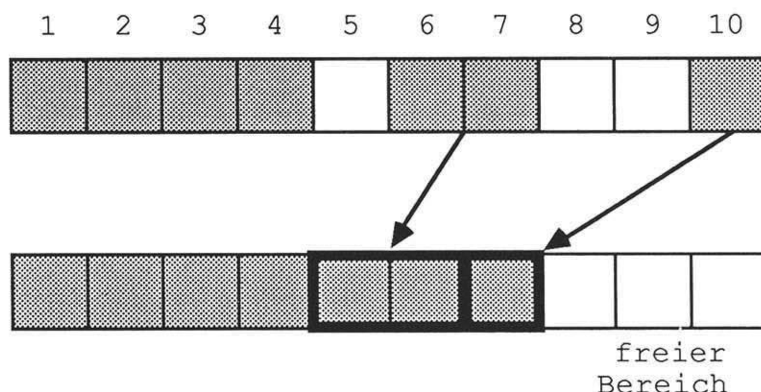


Bild 2 - 6: *Kompaktierung des Heaps*

Was passiert aber mit einem Zeiger, der auf einen der Blöcke verwies, die nach unten verschoben wurden? Ein Zeiger wird ja, wie schon oben erwähnt, üblicherweise als physikalische Adresse der Daten, auf die er verweist, gespeichert. Die Adresse dieser Daten hat sich jetzt geändert — nicht aber der Zeiger. Es wäre auch sehr schwierig, quasi "von Hand" unsere Zeiger-Variable zu ändern, so daß sie auf die alten Daten am neuen Ort zeigt. Diese Kompaktierung soll ja automatisch von der Speicherverwaltung durchgeführt werden, wenn sie Platz benötigt, um das eigentliche Programm von solchen Routine-Aufgaben zu entlasten. Wir merken deshalb gar nicht, wann diese Kompaktierung der belegten Speicherbereiche am unteren Ende des Heaps stattfindet, und können auch nicht darauf reagieren.

Deshalb ist das wahrscheinliche Ergebnis der Heap-Kompaktierung eine ganze Reihe von Zeigern, die ins Leere oder an die falschen Stellen zeigen. Man nennt das manchmal auch *baumelnde Zeiger* (auf engl. *dangling pointers*).

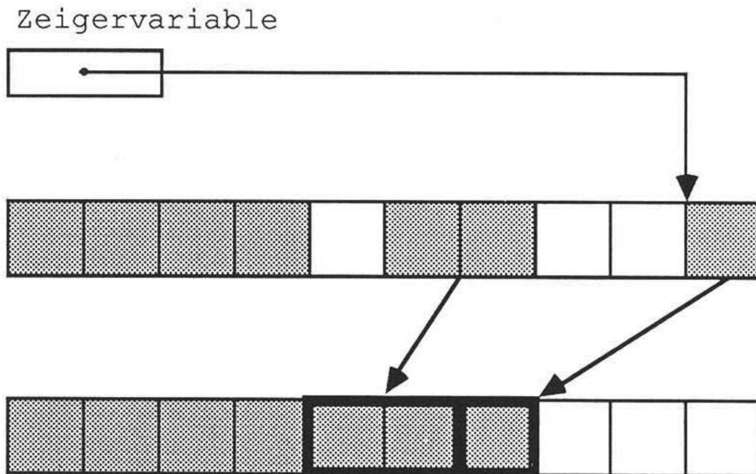


Bild 2 - 7: *Ein baumelnder Zeiger*

Eine Lösung dieses Problems könnte sein, der Speicherverwaltung zu jeder Variablen im Heap die Zeiger-Variable mitzuteilen, die darauf verweist. Die Speicherverwaltung könnte dann automatisch die Zeiger ändern, wenn sie die Daten verschiebt. Was aber, wenn mehrere Zeiger auf ein und dieselbe Heap-Variable zeigen, was gar nicht so selten vorkommt, wie man vielleicht denkt? Soll sich die Speicherverwaltung dann jeden dieser Zeiger merken? Daraus würde ein ziemlich hoher Speicherbedarf für diese "Rückverweise" resultieren. Außerdem müßte die Speicherverwaltung für jeden Block, den sie verschiebt, evtl. Hunderte von Zeigern ändern, was viel Zeit kosten würde.

Auf dem Macintosh hat man sich eine bessere Lösung einfallen lassen und die dafür nötigen Routinen auch in den ROM eingebaut. Wird ein neuer Speicherblock im Heap von der Speicherverwaltung angefordert, so kann vorher gewählt werden, ob man einen Zeiger darauf haben möchte oder einen Handle.

Wählt man den Zeiger, so wird ein neuer Block im Heap reserviert, als belegt markiert und dessen Adresse zurückgegeben. Wählt man einen Handle, wird

ebenfalls ein neuer Block angelegt, ein Zeiger auf diesen an einer anderen Stelle, im sogenannten Master-Pointer (MP), gespeichert und ein Zeiger auf den Master-Pointer zurückgegeben. Bei jedem Block, der im Heap reserviert wird, merkt sich die Speicherverwaltung zusätzlich noch einige Verwaltungsinformationen. So kann sie entscheiden, ob ein Zeiger oder ein Handle auf diese Daten zeigt, welcher Master-Pointer zu welchem Handle gehört und noch einiges mehr.

Muß der Heap jetzt kompaktiert werden, weil der Platz knapp wird, so können die Blöcke, die an einem Handle "hängen", beliebig verschoben werden. Die Speicherverwaltung ändert nach der Verschiebung nur den Master-Pointer entsprechend, und schon ist die Welt wieder in Ordnung. Selbst wenn mehrere Handles auf einen Block im Heap verweisen, braucht doch nur der eine gemeinsame Master-Pointer verändert werden.

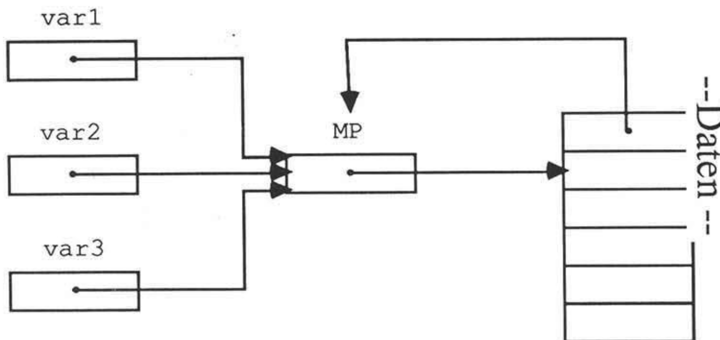


Bild 2 - 8: *Drei Handles an einem Master-Pointer*

Handles haben also enorme Vorteile. Sie benötigen zwar etwas mehr Speicherplatz für Verwaltungsaufgaben, sind dafür aber völlig frei im Speicher verschiebbar. Wenn Platz gebraucht wird, können solange Handles beiseite geräumt werden, bis ein genügend großer freier Block zusammenge - kommen ist.

Um an die Daten zu kommen, die an einem Handle hängen, müssen wir nun schon zwei Zeiger entlanglaufen. In Pascal schreibt man das in etwa wie folgt:

```
TYPE    IntPtr=          ^INTEGER;
        IntHandle=       ^IntPtr;
VAR
        handleVar:       IntHandle;
BEGIN
    ......
    handleVar^^ := 14;
    ......
END
```

Das Verfolgen zweier Zeiger (bzw. eines sog. *doppelt indirekten* Zeigers) kostet natürlich zusätzliche Zeit und auch etwas mehr Programmcode. Ein Opfer, das wir für die wesentlich erhöhte Flexibilität von Handles zahlen müssen.

Da es aber nur Handles ermöglichen, den Speicher des Macintosh optimal zu verwenden, kann man sie gar nicht oft genug einsetzen. Immer wenn es irgendwie möglich ist, sollte man sämtliche Daten, die nicht das ganze Programm über gebraucht werden, in dynamisch im Heap angelegte, verschiebbare Blöcke legen, auf die Handles verweisen. Leider verwendet die ToolBox selbst — aus historischen Gründen — an einigen Stellen einfache, nichtverschiebbare Blöcke. Wir werden diese Datenstrukturen in den folgenden Kapiteln noch kennenlernen. Wenn es irgendwie möglich ist, sollten all diese nichtverschiebbaren Blöcke — sofern wir sie das ganze Programm oder eine ganze größere Prozedur über brauchen — in den globalen oder loakalen Variablen liegen. Diese sind sowieso nichtverschiebbar, und es entstehen aus dieser Abspeicherung keine Nachteile. Im Heap sollten, wenn es irgendwie geht, nur verschiebbare Blöcke liegen.

Nichtverschiebbare Blöcke im Heap sind deshalb ein so großer Nachteil, weil sie bei jeder Heap-Kompaktierung "Inseln der Unbeweglichkeit" darstellen. Absicht einer Heap-Kompaktierung ist es ja gerade, sämtlichen freien Platz im einem Stück zu Verfügung zu haben. Und bei diesem Zusammenschieben von Speicherblöcken sind der Speicherverwaltung nichtverschiebbare Blöcke immer im Weg.

Einige Programme stürzen z.B. auf dem kleinen Mac mit 128K manchmal aus Speichermangel ab, obwohl im Heap (wie man mit den passenden Werkzeugen prüfen kann) noch einige tausend Bytes frei sind. Da aber mitten im Heap einige Datenstrukturen (z.B. von Fenstern, die aus historischen Gründen nichtverschiebbar sind) lagen, kann die Speicherverwaltung keinen freien Block von genügender Größe mehr finden und wirft eine Bombe.

2.6 Speicheraufteilung im Macintosh (MemoryMap)

Bis jetzt haben wir vier verschiedene Sorten von Variablen kennengelernt — globale, lokale und Zeiger-Variablen bzw. Handles, die auf eine im Heap angelegte dynamische Variable verweisen. Sinnvollerweise werden diese vier Gruppen von Variablen in verschiedenen Bereichen im Speicher angelegt, um ihre Verwaltung zu vereinfachen. Auf dem Macintosh geschieht das in drei verschiedenen Bereichen im Speicher. Die Anordnung dieser drei Bereiche sollte die Eigenschaften verschiedener Sorten von Variablen möglichst berücksichtigen. Die globalen Variablen benötigen z.B. nur einen festen Platz, während lokaleVariablen und erst recht dynamisch angelegte Variablen ganz unterschiedliche Speicherbedürfnisse im Laufe eines Programms haben können.

Bei vielen Computern, so auch auf dem Macintosh, wird dieses Wissen um die Eigenschaften der verschiedenen Variablen wie folgt genutzt: Ein Teil des gesamten Speicherbereichs des Computers wird fest für die Aufgaben des Betriebssystems, einen eventuell vorhandenen ROM, Bildschirmpuffer etc. vergeben. Die beim Programmstart bereits vergebenen Speicherbereiche liegen üblicherweise im oberen und/oder unteren Adreß-Bereich des verwendeten Prozessors. Den Rest (möglichst ein zusammenhängender Block) teilen sich der Heap und der Stack. Der Heap und der Stack liegen dabei zunächst einmal an entgegengesetzten Enden des Speichers und wachsen "gegeneinander an". Solange der Stack klein bleibt, kann deshalb der Heap sehr groß werden und umgekehrt. Der von einer Variablenart nicht genutzte Speicher steht der anderen voll zu Verfügung.

Auf dem Macintosh verwenden das Betriebssystem und die ToolBox sowohl einen großen Block am oberen Ende des Adreß-Bereichs wie auch einen kleinen Block am unteren Ende. Der Heap selbst wird schließlich noch einmal in zwei getrennte Heaps zerlegt, von denen der eine (kleinere) Teil für die Verwendung durch das Betriebssystem reserviert ist und der andere (weitaus größere) für das eigentliche Anwendungsprogramm und dessen Daten reserviert sind. Ja, Sie haben richtig gelesen: auch das Anwendungsprogramm liegt in Form eines oder mehrerer Datenblöcke im Heap.

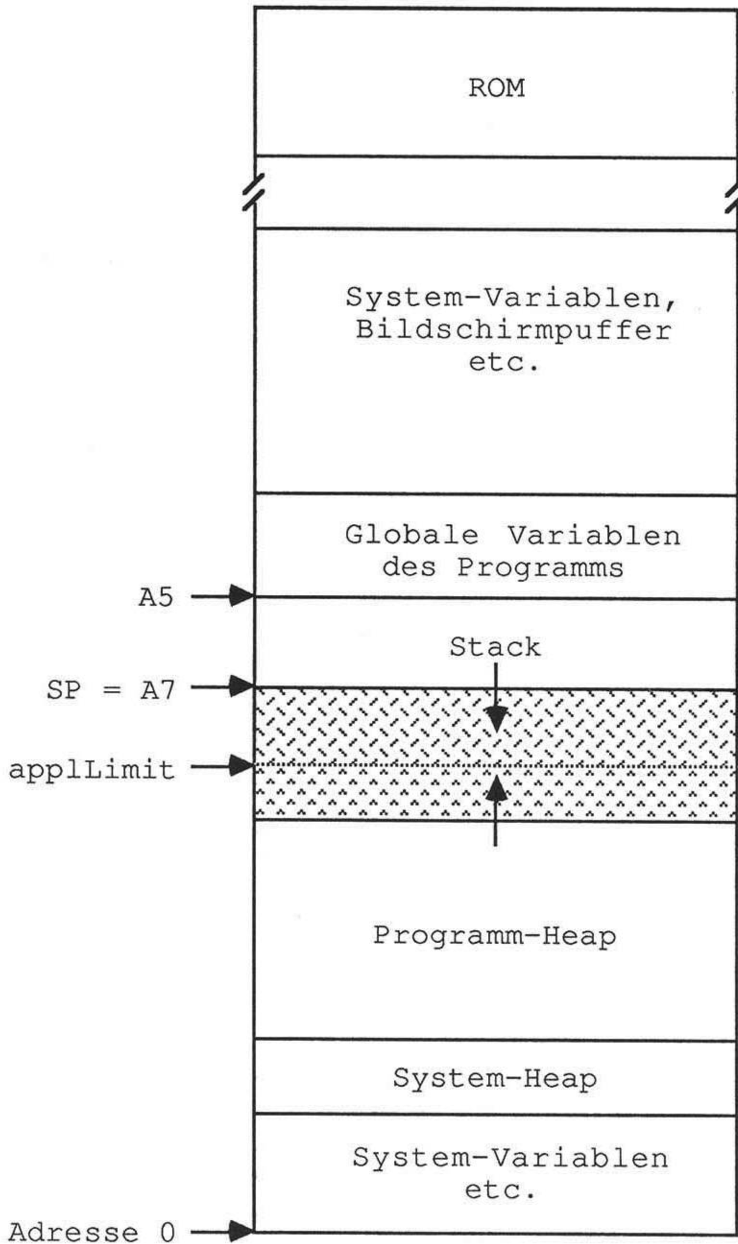


Bild 2 - 9: *Speicheraufteilung im Macintosh*

Wie in Bild 2 - 9 deutlich zu sehen ist, konkurrieren Stack und Heap auf dem Macintosh um den freien Speicher. Der Heap wächst nach oben, der Stack (aus technischen Gründen) nach unten. Auf dem Macintosh ist es dabei meistens so, daß der Heap sehr groß ist und fast alle Daten dort angelegt werden und der Stack relativ klein bleibt (weniger als ca. 8000 Byte). Erst wenn für beide nicht mehr genügend Platz im Speicher ist, kommt die berühmte Bombe. Dies kann entweder dadurch passieren, wenn ein neuer Platz im Heap angefordert wird, nicht mehr vorhanden ist und auch unter dem Stack nicht mehr genug Platz ist, um den Heap größer zu machen, oder wenn der Stack in den Heap hineinwächst (mit meist katastrophalen Folgen). Dies kann, wie wir oben gesehen haben, auch dann passieren, wenn eigentlich noch genug Platz im Heap frei ist — aber eben nicht genug an einem Stück.

2.7 Die Operationen der Speicherverwaltung des Macintosh

Die folgenden Prozeduraufrufe behandeln die Speicherverwaltung auf dem Macintosh. Diese Prozeduren sind alle bereits im ROM fest eingebaut und können problemlos von fast jeder Programmiersprache aufgerufen werden. Ich habe sie in ihrer Pascal-Form dargestellt, sie sieht aber in **C** und **Modula-2** meist genauso aus. In **Forth** und **Logo** lauten die Namen meist gleich, nur ist der Aufruf etwas anders.

2.7.1 Fehlerbehandlung

Alle Prozeduren der Speicherverwaltung tun ihr möglichstes, die Wünsche des Programms zu erfüllen. Wird z.B. ein (fester oder verschiebbarer) Block einer bestimmten Größe angefordert, so versucht der MemoryManager zunächst, einen freien Block dieser Größe zu finden. Ist keiner frei, wird der Heap kompaktiert wie oben beschrieben. Ist immer noch nicht genügend Platz frei, wird versucht, den Heap zu vergrößern, sowie einige andere Versuche angestellt, den freien Platz zu erweitern, die weiter unten erläutert werden. Erst wenn dann noch nicht genug Platz im Heap ist, gibt der MemoryManager auf und kehrt unverrichteter Dinge an das aufrufende Programm zurück. Vorher setzt er noch eine globale Variable des Betriebssystems auf einen Fehlercode, der das aufgetretene Problem charakterisiert.

Von Pascal aus kann dieser Fehlercode mit der Funktion **MemError** abgefragt werden. Von Assembler oder anderen Sprachen aus ist es meist möglich, diese Variable, die an einer festen Adresse liegt, direkt abzufragen.

```
FUNCTION MemError: OSErr;
    {OSErr ist in Pascal äquivalent zu INTEGER}
```

Die möglichen Werte von MemError sind die folgenden Konstanten:

```
CONST
    memFullErr        = -108; { Heap ist voll }
    memPurErr         = -112; { s.u.          }
    memWZErr          = -111; { dürfte nie passieren}
    nilHandleErr      = -109; { Operation mit leerem
                                Handle }
    noErr             = 0;    { Alles OK;kein Fehler}
```

Der einzige Fehler, mit dem wir es üblicherweise zu tun haben könnten, ist **memFullErr**, die anderen dürften nur in wirklich exotischen Situationen auftauchen.

2.7.2 Verwaltung von nichtverschiebbaren Blöcken

Es folgen nun die Prozeduren, mit denen "normale", nichtverschiebbare Blöcke von Daten im Heap verwaltet werden können.

```
FUNCTION NewPtr (byteCount: Size): Ptr;
```

NewPtr reserviert einen Block von **byteCount** Bytes im Heap und gibt einen Zeiger darauf zurück. Falls auch nach einem eventuellen Zusammenschieben des Heaps nicht **byteCount** Bytes in einem Stück frei sind, gibt es einen Fehler, und der Zeiger **NIL** wird zurückgeliefert.

```
PROCEDURE DisposPtr (p: Ptr);
```

Die Prozedur **DisposPtr** gibt den Speicherblock im Heap, auf den **p** zeigt, wieder frei (markiert ihn als wiederverwendbar). Danach sollte auf keinen Fall versucht werden, mit den Daten, auf die dieser Zeiger verwies, noch irgend etwas zu unternehmen. **DisposPtr** liefert den Fehlercode **memWZErr**, wenn **p** auf einen freien Block im Heap verwies.

```
FUNCTION GetPtrSize (p: Ptr): Size;
```

GetPtrSize stellt die Größe in Bytes des Speicherblocks im Heap fest, auf den **p** verweist. Der Typ **Size** ist in Pascal äquivalent zu **LONGINT** (eine 32 Bits große ganze Zahl).

```
PROCEDURE SetPtrSize(p: Ptr; newSize: Size);
```

SetPtrSize verändert die Größe des Speicherblocks im Heap, auf den **p** verweist, sofern dies möglich ist. Wenn der Block größer werden soll und unmittelbar dahinter bereits ein nichtverschiebbarer Block liegt oder der Heap zu Ende ist, gibt es natürlich den Fehler **memFullErr**.

2.7.3 Verwaltung von verschiebbaren Blöcken

Nun folgen die Prozeduren, die mit Handles zu tun haben. Hierzu allerdings zunächst noch ein paar Informationen:

Mit jedem Block, auf den ein Handle verweist, merkt sich die Speicherverwaltung noch einige zusätzliche Statusinformationen zu diesem Block. Da ist zunächst einmal natürlich der Rückverweis auf den entsprechenden Master-Pointer und die Information, daß es sich überhaupt um einen verschiebbaren Block an einem Handle handelt. Dazu kommen aber noch zwei Flags namens **locked** (deutsch ungefähr gleich "festgenagelt") und **purgeable** (ungefähr gleich "löschbar").

Ist ein Block **locked**, so kann er, obwohl er an einem Handle hängt, nicht bewegt werden. Dies ist meist ein vorübergehender Zustand. Spielt z.B. die Geschwindigkeit eine sehr große Rolle, so ist es oft sinnvoll, einen Block zu **locken**, sich einen Zeiger direkt auf den Speicherblock im Heap zu holen und mit diesem weiterzuarbeiten. Man sollte das **locked**-Flag aber so bald wie möglich wieder zurücksetzen.

Das **purgeable**-Flag ist nur für fortgeschrittenere Programmierer bzw. trickreiche Programme interessant. Es wird insbesondere im Zusammenhang mit der Resource-Verwaltung des Macintosh verwendet, die in einem späteren Kapitel noch näher erläutert wird. Ist ein Block purgeable (ist das **purgeable**-Flag gesetzt), so kann dieser Block, wenn es wirklich eng im Heap wird, gelöscht werden. Dies ist in den wenigsten Fällen wünschens - wert; also Vorsicht!

Nun zu den eigentlichen Operationen mit Handles:

```
FUNCTION NewHandle (byteCount: Size): Handle;
```

Diese Funktion reserviert einen Block von **byteCount** Bytes im Heap, legt einen Master-Pointer dafür an und gibt einen Zeiger auf den Master-Pointer zurück. Falls auch nach einem eventuellen Zusammenschieben des Heaps nicht **byteCount** Bytes in einem Stück frei sind, gibt dies den Fehler **memFullErr**.

```
PROCEDURE DisposHandle (h: Handle);
```

DisposHandle gibt den Speicherblock im Heap, auf den **h** zeigt, wieder frei (markiert ihn als wiederverwendbar) und löscht auch den Master-Pointer. Über keine der Handle-Variablen, mit denen auf diesen Block verwiesen wurde (es können ja viele sein), darf danach versucht werden, auf die Daten in diesem Block zuzugreifen!

```
FUNCTION GetHandleSize (h: Handle): Size;
```

Diese Funktion liefert die Größe in Bytes des Speicherblocks im Heap zurück, auf den **h** verweist.

```
PROCEDURE SetHandleSize(h: Handle; newSize: Size);
```

SetHandleSize versucht, die Größe des Speicherblocks im Heap, auf den **h** verweist, auf **newSize** Bytes zu verändern. Wenn nicht mehr genügend Bytes im Heap frei sind, gibt es natürlich den Fehler **memFullErr**.

Die folgenden Operationen dienen zum Setzen und Löschen der beiden Flags **locked** und **purgeable**. Ihre Funktion dürfte relativ klar sein.

```
PROCEDURE HLock(h: Handle);
PROCEDURE HUnLock(h: Handle);
PROCEDURE HPurge(h: Handle);
PROCEDURE HNoPurge(h: Handle);
```

2.7.4 Operationen für den Heap als Ganzes

Der MemoryManager bietet auch einige Operationen an, mit denen der Heap als Ganzes modifiziert werden kann bzw. Informationen darüber gewonnen werden können.

```
PROCEDURE SetApplLimit(zoneLimit: Ptr);
```

SetApplLimit setzt dem Heap eine Grenze (engl. *limit*), bis zu der er sich maximal ausdehnen kann. **zoneLimit** muß dazu die Adresse des ersten Bytes hinter dieser Grenze enthalten. Wenn der Heap bereits größer geworden ist, wird er dadurch nicht kleiner, hört aber auf alle Fälle auf, sich auszudehnen. Standardmäßig ist die Grenze des Heap-Wachstums bereits auf ca. 8000 Bytes unter dem Beginn des Stacks festgelegt. Davon sollte nur in seltenen Fällen abgewichen werden, z.B. wenn ein Programm durch rekursive Prozeduren extrem viel Stack benötigt. Zudem kann es in anderen Sprachen als LisaPascal oder Assembler gefährlich sein, den voreingestellten Wert zu ändern, oder es gibt andere Wege, auf denen man dies tun muß. Prüfen Sie auf alle Fälle erst die Dokumentation ihrer Programmier-Sprache, bevor Sie **SetApplLimit** anwenden.

```
PROCEDURE MoreMasters;
```

MoreMasters ist eine sehr wichtige Operation, die zu Beginn jedes Programm aufgerufen werden sollte (prüfen Sie aber zunächst, was die Dokumentation Ihrer Programmiersprache dazu sagt). **MoreMasters** legt einen Block von (üblicherweise 64) MasterPointern für Handles im Heap an. Da auf MasterPointer ja mit einfachen Zeigern verwiesen wird, sind sie natürlich nicht verschiebbar und bilden deshalb Inseln im Heap. Der MemoryManager selbst ruft immer dann **MoreMasters** auf, wenn er einen neuen Handle anlegen muß und keinen MasterPointer mehr frei hat. **MoreMasters** schafft gleich 64 Stück davon auf Vorrat. **DisposHandle** gibt auch den MasterPointer des freigegebenen Handles frei, der danach für den nächsten **NewHandle**-Aufruf wiederverwendet werden kann.

MoreMasters ruft der MemoryManager also wirklich nur dann auf, wenn kein MasterPointer mehr frei ist.

Wenn gleich am Anfang eines Programms genügend MasterPointer angelegt werden, um jedem Handle, das im Programm verwendet wird, einen zu garantieren, wird das automatische Anlegen von MasterPointern durch den MemoryManager verhindert und damit natürlich auch das Bilden einer nichtverschiebbaren Insel im Heap.

```
FUNCTION FreeMem: LONGINT;
```

Die Funktion **FreeMem** gibt die Anzahl aller noch freien Bytes im Heap an. Eine eventuelle Ausdehnung des Heaps wird dabei nicht berücksichtigt. Normalerweise ist es allerdings nie möglich, einen Block der Größe **FreeMem** Bytes anzufordern, da nichtverschiebbare Blöcke im Heap dies verhindern. Gibt es keine nichtverschiebbaren Blöcke oder liegen diese alle am unteren Ende des Heaps, kann allerdings wirklich ein Block dieser Größe angefordert werden.

```
FUNCTION MaxMem(VAR grow: Size): Size;
```

MaxMem tut alles, was in der Macht des MemoryManagers steht, um im Heap Platz zu schaffen, und liefert die Größe des größten dann freibleibenden Blocks im Heap als Funktionsergebnis zurück. Der VAR-Parameter **grow** wird auf die Anzahl Bytes gesetzt, um die der Heap noch wachsen könnte, bevor er an seine Grenze stößt.

2.7.5 Hilfsoperationen der Speicherverwaltung

Eine Reihe von Operationen, die den Umgang mit Handles und Pointern vereinfachen, sind bereits in die ToolBox eingebaut. Sie sind zwar in *Inside Macintosh* im Kapitel *Operating System Utilities* erläutert, gehören logisch aber zur Speicherverwaltung. Diese Operationen liefern als Funktionswert meist alle schon den Fehlercode zurück, den man bei den oben erklärten Operationen mit **MemErr** abfragen mußte.

```
FUNCTION HandToHand(VAR: theHandle: Handle)
          : OSErr;
```

HandToHand legt eine Kopie der Daten, auf die der Handle **theHandle** verweist, im Heap an und liefert einen Handle auf diese Kopie im VAR-Parameter **theHandle** zurück. Die alten Daten liegen nach wie vor im Heap,

nur können sie nun über die Variable, die man als Parameter **theHandle** übergeben hat, nicht mehr angesprochen werden. Falls nicht genug Platz im Heap ist, tritt natürlich der Fehler **memFullErr** auf.

```
FUNCTION PtrToHand
             (srcPtr:         Ptr;
              VAR theHandle: Handle;
              size:           LONGINT
             ): OSErr;
```

PtrToHand legt ähnlich wie **NewPtr** einen neuen verschiebbaren Block der Größe **size** im Heap an, auf den ein Handle im VAR-Parameter **theHandle** zurückgeliefert wird. In diesen Block werden aber sofort die Daten, auf die **srcPtr** zeigt, kopiert.

```
FUNCTION PtrToXHand
             (srcPtr:     Ptr;
              theHandle: Handle;
              size:       LONGINT
             ): OSErr;
```

PtrToXHand arbeitet ähnlich wie **PtrToHand**. Nur muß der Parameter **theHandle** auf einen bereits im Heap existierenden verschiebbaren Block verweisen, dessen Größe von **PtrToXHand** auf **size** gesetzt wird. In diesen Block werden die Daten, auf die **srcPtr** zeigt, kopiert.

```
FUNCTION HandAndHand(aHandle,bHandle: Handle)
             : OSErr;
```

HandAndHand hängt eine Kopie des verschiebbaren Speicherblocks, auf den **aHandle** verweist, hinten an den Block, auf den **bHandle** verweist, an. Vorher wird der Block, auf den **bHandle** verweist, natürlich um die Größe von **aHandle** verlängert, um die neuen Daten aufnehmen zu können.

```
FUNCTION PtrAndHand
             (srcPtr:     Ptr;
              theHandle: Handle;
              size:       LONGINT
             ): OSErr;
```

PtrAndHand hängt die Daten, auf die **srcPtr** zeigt, hinten an den verschiebbaren Speicherblock, auf den **theHandle** verweist, an. Dieser

Block wird vorher natürlich um **size** Bytes vergrößert, damit er die neuen Daten aufnehmen kann.

```
PROCEDURE BlockMove(src,dst: Ptr; numBytes: Size);
```

BlockMove ist die primitivste Operation des MemoryManagers. Sie bewegt einen Block der Länge **numBytes** von der Adresse **src** zur Adresse **dst**. Diese Prozedur führt keinerlei Überprüfungen mit ihren Parametern durch. Man kann sich damit herrlich sein gesamtes Programm zerlegen. Bei korrekter Anwendung arbeitet **BlockMove** allerdings sehr schnell und recht intelligent. So werden z.B. auch Fälle, bei denen sich der Quell- und Ziel-Bereich überschneiden, stets korrekt behandelt.

2.8 Vorsicht, Handles! (Gefahren von Handles)

Handles bringen bei all ihren Vorteilen eine Reihe von Problemen mit sich. Einige dieser Probleme tauchen nur in Pascal und Modula-2 auf und sind zudem abhängig vom verwendeten Compiler. Es ist aber auf alle Fälle sicherer, die beschriebenen Problemfälle auch dann zu umschiffen, wenn man eine andere Sprache verwendet. Gerade die Eigenschaften von Handles führen oft zu Programmfehlern, die sehr schwer zu finden sind, weil sie in Programmstücken sind, die vollkommen korrekt aussehen — und es auch wären, wenn die Daten an Handles nicht verschiebbar wären. Zudem können diese Konstruktionen manchmal funktionieren und manchmal nicht, oder erst dann nicht mehr, wenn man an einer ganz anderen Stelle im Programm etwas geändert hat.

Pascal und Modula-2 haben die unangenehme Eigenschaft, es dem Programmierer kaum zu zeigen, wann eine Adresse berechnet wird. Übergibt man z.B. eine Variable als Parameter an eine Prozedur, so kann es sein, daß dieser Prozedur der Wert der Variablen übergeben wird, oder ihre Adresse, damit der Wert von dieser Prozedur verändert werden kann. Dies ist dem Aufruf der Prozedur aber nicht ansehbar; man muß bei der Definition der Prozedur nachschauen, ob der bewußte Parameter ein VAR-Parameter ist.

```
PROCEDURE X (param: INTEGER);
BEGIN
    ......
    param := 4; { keine Wirkung außerhalb von X }
    ......
END; { X }
```

```
PROCEDURE Y (VAR param: INTEGER);
BEGIN
    ......
    param := 4; { Wirkung auch außerhalb von Y }
    ......
END; { X }

VAR
    a:        INTEGER;
    ......
    X(a);         { Wert     von a wird an X übergeben }
    Y(a);         { Adresse von a wird an Y übergeben }
```

Liegt der übergebene Wert für den Parameter aber in einem verschiebbaren Speicherblock, wird die ganze Sache kritisch. Die Adresse des Parameters wird vor dem Aufruf der Prozedur berechnet und an die Prozedur übergeben, die mit dieser Adresse arbeitet, als stände bei dieser Adresse im Speicher der Wert einer Variablen des richtigen Typs. Diese Annahme ist auch nicht so abwegig und "normalerweise" immer richtig — nur eben nicht bei Handles. Schauen wir uns dazu einmal das folgende Programmstück an:

```
TYPE
    IntPtr        = ^INTEGER;
    IntHandle     = ^IntPtr;

VAR
    a:        IntHandle;
    ......
    a    := IntHandle(NewHandle(2));
                        { INTEGER ist 2 Bytes groß }
    a^^  := 15;
    X(a^^);             { vollkommen ungefährlich  }
    Y(a^^);             { SEHR GEFÄHRLICH }
```

Wir erzeugen zu Beginn des kleinen Programmstücks einen neuen verschiebbaren Block der Größe 2 im Heap, auf den **a** verweist. (Das Funktionsergebnis von **NewHandle** müssen wir wegen des Typschutzes von Pascal erst in den korrekten Typ umwandeln. In LisaPascal oder Modula-2 geht das, wie hier gezeigt, in anderen Sprachen muß man eventuell andere Konstruktionen verwenden, oder eine Umwandlung ist gar nicht

nötig.) In diesen Block schreiben wir dann die Zahl 15 und rufen mit diesem neuen Wert die Prozeduren **X** und **Y**, die uns ja bereits bekannt sind, auf.

Bei **X** bedeutet dies keine Probleme, allerdings kann der Inhalt des Speicherblocks durch die Prozedur auch nicht geändert werden. Der Aufruf von **Y** kann sofort oder später zu einer Bombe führen!

Sobald nämlich in **Y** irgendeine Operation durchgeführt wird, die den Heap kompaktiert, wird der Datenblock, auf den **a** verweist, wahrscheinlich verschoben. Er befindet sich wahrscheinlich nicht mehr bei der Adresse, die der Compiler beim Aufruf **Y** berechnet hat. Sobald in **Y param** ein neuer Wert zugewiesen wird, wird dieser Wert aber in eine Speicherstelle bei der alten Adresse geschrieben. Und wer weiß, was für wertvolle andere Daten sich inzwischen (durch die Heap-Kompaktierung) dort befinden.

Man kann natürlich darauf achten, wann es möglich ist, daß der Heap kompaktiert wird, und wann nicht. Man übersieht so etwas aber schnell oder vergißt es, wenn man später Änderungen am Programm durchführt. In einigen Programmiersprachen kann sogar der Aufruf einer Prozedur zu einer Heap-Kompaktierung führen, deshalb sollte man mit Handles und VAR-Parametern sehr vorsichtig umgehen. Dieses Problem wird in Pascal und Modula dadurch verschärft, daß einem Prozedur-Aufruf nicht anzusehen ist, welcher Parameter als Wert und welcher als Adresse übergeben wird. Aber auch in allen anderen Programmier-Sprachen sind solche Fehler möglich. Sie können nur im Programmtext leichter indentifiziert werden. Prinzipiell ist immer sicherer, solche Konstruktionen zu vermeiden oder abzusichern, wie das folgende Programmstück zeigt:

```
VAR
    a:        IntHandle;
    b:        INTEGER;
    ......
    a   := IntHandle(NewHandle(2));
    a^^ := 15;
    ......
    b := a^^;              { 1. sichere Möglichkeit }
    Y(b);
    a^^:= b;
    --
    HLock(Handle(a));      { 2. sichere Möglichkeit }
    Y(a^^);
    HUnLock(Handle(a));
```

Im ersten Fall weisen wir den Inhalt des verschiebbaren Speicherblocks einer Hilfsvariablen zu, mit der wir dann den Prozedur-Aufruf durchführen. Nach dem Aufruf von **Y** muß der Wert der Hilfsvariablen natürlich wieder in den verschiebbaren Speicherblock befördert werden. Die zweite Möglichkeit ist, den verschiebbaren Speicherblock durch den Aufruf von **HLock** vor dem Aufruf von **Y** im Heap festzusetzen und sofort danach wieder zu lösen. Wie oben bereits beschrieben, macht **HLock** aus einem verschiebbaren einen nichtverschiebbaren Block im Heap, und **HUnlock** mach ihn wieder verschiebbar.

Eine ähnlich gefährliche Konstruktion ist die Zuweisung eines Funktionsergebnisses an einen verschiebbaren Block. Dies ist zwar stark vom verwendeten Compiler bzw. Interpreter abhängig, sollte aber sicherheitshalber immer vermieden werden.

```
FUNCTION Z: INTEGER;
BEGIN
END; { Z }
......
VAR
    a:          IntHandle;
    b:          INTEGER;
    ......
    a   := IntHandle(NewHandle(2));
    a^^ := 15;
    ......
    a^^ := Z;               { GEFÄHRLICH }
    ......
    b := a^^;               { 1. sichere Möglichkeit }
    b := Z;
    a^^:= b;
    ......
    HLock(Handle(a));       { 2. sichere Möglichkeit }
    a^^ := Z;
    HUnLock(Handle(a));
```

Die dritte Gefahrenstelle im Zusammenhang mit Handles lauert bei dem so praktischen WITH-Statement in Pascal und Modula-2, das auch wieder besonderer Absicherung bedarf. Ähnliche Konstruktionen gibt es in anderen Sprachen kaum. Ein WITH-Statement dient der vereinfachten Handhabung von RECORD-Variablen und spart sowohl Typarbeit wie üblicherweise auch Programmcode und Laufzeit ein.

```
TYPE
    DreiInts=
    RECORD
        x,y,z: INTEGER;
    END; { DreiInts }
......
VAR
    a:      DreiInts;
    ...........
    a.x := 3;                    { umständlich }
    a.y := 5 + a.x;
    a.z := (a.x + a.y) DIV 2;
    ......
    WITH a DO  { Compiler berechnet Adresse von a }
        BEGIN  { weniger Schreibarbeit }
        x := 3;
        y := 4+x;
        z := (x+y) DIV 2;
        END; {WITH a}
```

Bei dem Befehl **WITH a** berechnet der Compiler einmal die Adresse von **a** und merkt sie sich nach Möglichkeit in einem Register. Um danach die Adressen der einzelnen Felder zu berechnen, wird immer deren Abstand (*Offset*) vom Beginn eines Records vom Typ **DreiInts** auf dies Register aufaddiert. Die komplette Berechnung der Adresse von **a** selbst unterbleibt und spart Zeit und Code. Diese Berechnung ist bei globalen Variablen nicht aufwendig, kann aber bei lokalen Variablen oder bei Variablen, die über lange Zeigerkette erreicht werden (z.B. **WITH rec1.b^.wert^.vorgaenger**), eine Menge Code und Zeit sparen.

Liegt der Record, auf den das WITH-Statement zugreift, jedoch in einem verschiebbaren Speicherblock, wird es wieder gefährlich. Die beim Einritt in das WITH-Statement berechnete Adresse des Beginns der Record-Variablen kann bei der ersten Heap-Kompaktierung ungültig werden. Besonders gefährlich wird es, wenn innerhalb des WITH-Statements andere Prozeduren oder Funktionen aufgerufen werden. Die Absicherung in solchen Fällen heißt wieder Festnageln der Speicherblocks vor Beginn des gefährlichen Gebietes und Lösen nach Verlassen dieses Gebietes:

```
TYPE
    DreiIntPtr= ^DreiInts;
    DreiIntHdl= ^DreiIntPtr;
VAR
    a:       DreiIntHdl;

    a    := DreiIntHdl(NewHandle(SIZEOF(6)));
              { Drei INTEGER sind 6 Bytes groß }
    ......
    WITH a^^ DO { GEFÄHRLICH }
        BEGIN
        x := 3;
        y := FUNKTION(4+x);
        z := (x+y) DIV 2;
        END; {WITH a}

    ......
    HLock(Handle(a));      { sichere Möglichkeit }
    WITH a^^ DO
        BEGIN
        x := 3;
        y := FUNKTION(4+x);
        z := (x+y) DIV 2;
        END; {WITH a^^}
    HUnLock(Handle(a));
```

3 QuickDraw

Ein großer Teil des ROMs im Macintosh wird vom Grafikpaket "QuickDraw" gefüllt. QuickDraw (zu deutsch: "SchnellZeichner") ist ein Paket von Grafikroutinen, das an Vielseitigkeit und Geschwindigkeit seinesgleichen sucht. In seiner jetzigen Form ist es direkt in Assembler geschrieben worden, der "Muttersprache" des Macintosh. Es hat seine Ursprünge in einem in Pascal geschriebenen Grafikpaket gleichen Namens auf der Lisa. Während QuickDraw auf der Lisa aber noch eine Größe von über 150.000 Byte hatte, ist es jetzt im Macintosh nur noch ca. 23.000 Byte groß und zudem noch viel schneller. Übrigens ist QuickDraw auch einer der Hauptgründe, weshalb der Macintosh einen 68000-Prozessor bekommen hat. Zunächst war nur an die Verwendung des viel preiswerteren 6809 8-Bit-Prozessors gedacht worden. Dem Designteam des Macintosh gefielen QuickDraw und die damit eng zusammenhängende Fensterverwaltung der Lisa aber so gut, daß man sich entschloß, denselben Prozessor wie die Lisa — eben einen MC68000 — zu verwenden, um nicht noch einmal ein solches Paket für den 6809 schreiben zu müssen. Dieses hätte wahrscheinlich einige Mannjahre gekostet und wäre bei weitem nicht so leistungsfähig geworden.

Eine gute Kenntnis von QuickDraw ist für jeden, der lernen will, wie man den Macintosh programmiert, eine Grundvoraussetzung. Alle Ausgaben, die ein Programmierer dem Anwender auf dem Bildschirm des Macintosh zeigen will, laufen über Quickdraw. Dies gilt nicht nur für Grafikprogramme wie MacPaint oder MacDraw, sondern für jedes Programm, das irgend etwas auf dem Bildschirm darstellen möchte. Hinzu kommt noch, daß eine gesunde Grundkenntnis von QuickDraw unbedingt notwendig für das Verstehen von Fenstern ist. Deshalb habe ich dieses Kapitel auch ziemlich an den Anfang dieses Buches gesetzt, obwohl es vielleicht etwas trocken ist.

Um die Unterschiede und Zusammenhänge zwischen den mathematischen Konzepten, die QuickDraw zugrunde liegen, und dem konkreten Zeichnen auf dem Bildschirm zu verdeutlichen, werde ich zunächst eine kleine Einführung in die mathematischen Grundlagen geben und danach die Verbindung zur Hardware aufzeigen.

3.1 Mathematische Grundlagen von QuickDraw

Die Arbeit mit QuickDraw und das Verständnis dieses komplexen und mächtigen Grafikpakets wird viel einfacher, wenn man zunächst die mathematischen Konzepte verstanden hat, auf denen QuickDraw basiert. QuickDraw ist ein Grafikpaket für sogenannte "bitmapped" Grafik. Das heißt, daß die Video-Hardware des Macintosh eine bestimmte Menge von Bits im Speicher auf dem Bildschirm abbildet. Jedes Bit entspricht einem Punkt auf dem Bildschirm. Ist dieses Bit gleich 1, so ist der entsprechende Punkt schwarz. Ist es gleich 0, so ist der Punkt weiß. (QuickDraw ist zwar bereits für Farbdarstellung vorbereitet, wobei für jeden farbigen Punkt mehrere Bits verwendet werden, alle im Moment erhältlichen Computer der Macintosh-Serie kennen aber nur schwarz/weiße Darstellung.)

Anders als bitmapped Grafikpakete auf anderen Computern ist QuickDraw aber nicht sklavisch abhängig von der Darstellung einer Grafik am Bildschirm. Alle Datenstrukturen und Operationen sind zunächst für "mathematisch reine" Objekte wie Punkte, Rechtecke und Flächen definiert und werden erst später mit bestimmten Bits im Speicher bzw. Punkten am Bildschirm verbunden. Dies hat eine ganze Reihe von Vorteilen, auf die ich an dieser Stelle nicht näher eingehen kann, verlangt aber zunächst einmal den Einsatz von etwas "Gehirnschmalz", da man nicht sofort mit Malen von Strichen und Kreisen beginnen kann, ohne sich darüber einige Gedanken zu machen.

3.1.1 Die virtuelle Grafikebene

Alle QuickDraw-Operationen spielen sich in einer virtuellen Grafikebene zwischen den Koordinaten –32768 und 32767 horizontal (auf der X-Achse) und –32768 und 32767 vertikal (auf der Y-Achse) ab. Die Werte auf der Y-Achse werden — im Gegensatz zur uns aus der Schule vertrauten Geometrie — nach unten größer.

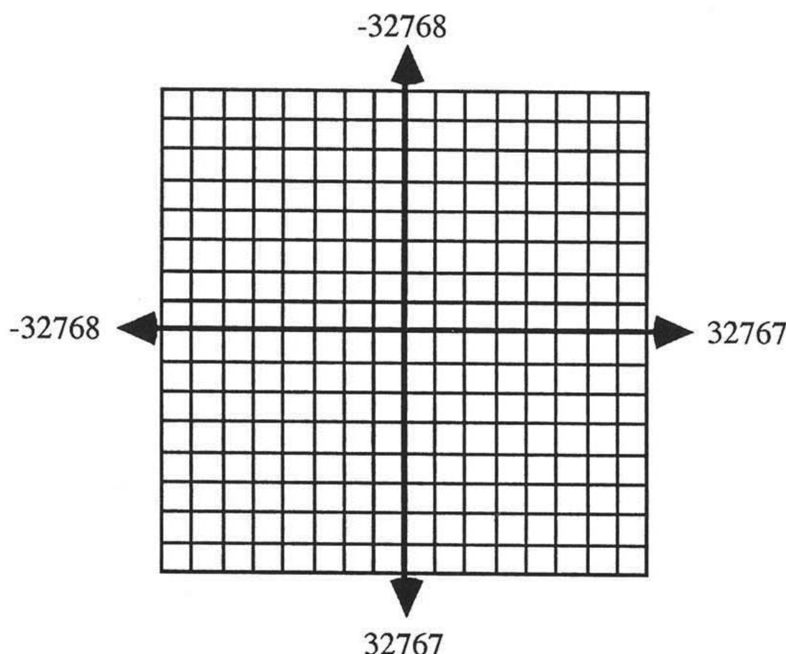


Bild 3 - 1:*Die Grafikebene von QuickDraw*

Auf dieser riesigen Ebene befinden sich 4 294 967 296 einzelne "QuickDraw-Punkte" — Kreuzungspunkte der vertikalen und horizontalen Linien in der obigen Abbildung. Auf der ganzen Ebene können Zeichnungen mit QuickDraw-Operationen dargestellt werden. Der Bildschirm zeigt allerdings immer nur einen relativ kleinen Ausschnitt davon.

Auf dieser Ebene existieren die Objekte, die QuickDraw für fast alle Zeichen-Operationen benötigt: Punkte, Rechtecke und Regionen.

3.1.2 Punkte

Ein Punkt kann in einer Variablen vom Typ **Point** gespeichert werden. Der Typ **Point** ist dabei in Pascal definiert wie folgt:

```
TYPE    VHSelect=(V,H);
```

```
Point = RECORD CASE Integer OF
   0:   (v:     Integer ;
         h: Integer );

   1:   (vh: ARRAY [VHSelect] OF Integer

        END {OF RECORD Point};
```

Eine Variable vom Typ Point ist also ein sog. "varianter Record". Nehmen wir an, wir hätten in unserem Programm eine Varaible namens **EinPunkt** vereinbart, dann könnten wir die horizontale Koordinate dieses Punktes sowohl als **EinPunkt.h**, wie auch als **EinPunkt.vh[H]** ansprechen. Da beide Auffassungsweisen gleich gespeichert werden -- als zwei hintereinanderliegende Integer — kann man ohne Probleme zwischen ihnen hin und her wechseln. Diese doppelte Sichtweise kann für viele Anwendungen recht praktisch sein und wird uns noch oft begegnen.

Wichtig bei den QuickDraw-Punkten ist, daß es sich nicht um Punkte auf dem Bildschirm handelt, sondern um abstrakte Konzepte. Wie QuickDraw Bildschirm-Punkte auffaßt, werden wir später noch sehen.

3.1.3 Rechtecke

Für viele Operationen von QuickDraw werden auch Rechtecke benötigt. Ein Rechteck wird vollständig beschrieben durch 4 Zahlen (linker, rechter, oberer und unterer Rand) oder 2 Punkte (linke, obere Ecke und rechte, untere Ecke). Entsprechend sieht auch der Typ für solche Variablen in Pascal aus:

```
TYPE Rect = RECORD  CASE Integer OF
            0:    ( top:     Integer ;
                    left:    Integer ;
                    bottom:  Integer ;
                    right:   Integer );

            1:  ( topLeft:  Point;
                  botRight: Point);
            END {OF RECORD Rect };
```

Bild 3 - 2 zeigt das Rechteck ((2,5),(10,10)), das durch zwei Punkte (2,5) und (10,10) beschrieben wird. Zu diesem Rechteck gehört die gesamte schraffierte Fläche zwischen den etwas dicker gedruckten Linien.

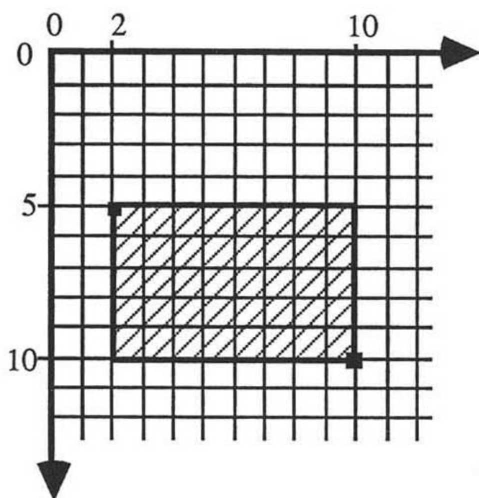


Bild 3 - 2:*Ein Reckteck*

Rechtecke sind allerdings — genau wie Punkte — zunächst einmal rein mathematische Objekte. Eine Zuordnung von Rechtecken zu einer bestimmten Darstellung am Bildschirm erfolgt erst durch die sogenannte BitMap, zu der ich später noch kommen möchte.

3.1.4 Regionen

Eine sehr wichtige Konstruktion für alle Grafikanwendungen und besonders auch für die Verwaltung von Fenstern, zu denen wir im nächsten Kapitel kommen werden, sind Regionen. Eine Region ist ein beliebiger Bereich auf der Grafikebene. Seine Form braucht weder rechteckig noch rund, noch irgendwie durch eine andere mathematische Funktion bestimmbar sein.

Man könnte auch sagen, daß eine Region eine beliebige Menge der kleinen weißen Kästchen zwischen den Koordinatenlinien der Grafikebene ist. Diese Kästchen sind nämlich die kleinste Flächeneinheit, die QuickDraw kennt, da es ja mit ganzen Zahlen arbeitet.

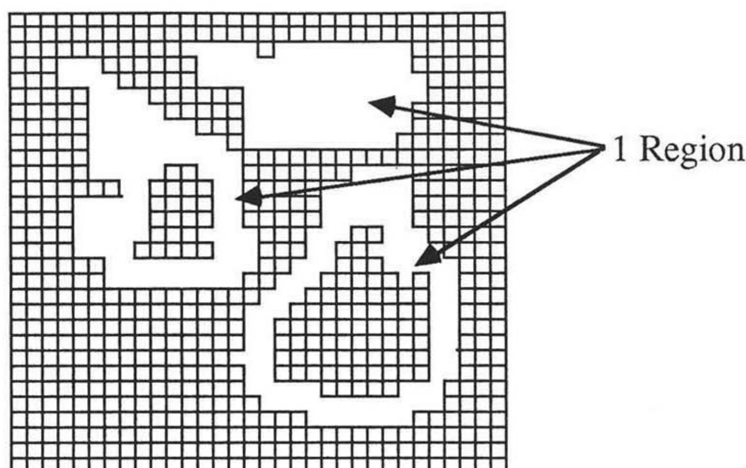


Bild 3 - 3: *Eine Region*

Die Datenstruktur, mit der eine solche Region beschrieben werden kann, ist natürlich stark abhängig von ihrer genauen Form. Auch die Größe dieser Datenstruktur schwankt mit der Komplexität dieser Form. Eine solche Datenstruktur läßt sich in Pascal und auch in anderen Programmiersprachen deshalb nicht genau beschreiben. In *Inside Macintosh* wird sie definiert, wie folgt:

```
TYPE Region =
      RECORD
        rgnSize:      Integer ;
        rgnBox:       Rec;
        rgnData:       ARRAY [0..32000] OF Byte;
      END {OF RECORD Region};
```

rgnSize ist die wirkliche Länge des Records. Um Speicher zu sparen, ist **rgnData** nämlich immer nur so groß, wie gerade für diese Region nötig. **rgnBox** ist das kleinste Rechteck, das die Region gerade noch vollständig umschließt. Der interessante Teil verbirgt sich natürlich im Array **rgnData**, das hier nur als amorphe Kette von Bytes definiert ist, in Wirklichkeit aber eine viel kompliziertere Struktur hat.

Da jede Änderung der Region auch eine Änderung der Größe ihrer Beschreibung mit sich bringen kann, wird sie normalerweise als variabler, verschiebbarer Speicherblock verwaltet:

```
TYPE RgnPtr =          ^Region;
     RgnHandle =       ^RgnPtr;
```

Solche "Handles" kennen wir ja schon aus dem letzten Kapitel. Ihr Hauptvorteil besteht eben darin, daß die Speicherverwaltung des Macintosh die Daten, die "an einem Handle hängen", beliebig größer und kleiner machen und auch im Speicher verschieben kann, ohne daß auch nur ein Byte vergeudet wird oder das Programm etwas davon merkt.

Nun aber vorläufig genug von abstrakten Konzepten. Betrachten wir jetzt einmal, mit welchen Hardwarevoraussetzungen QuickDraw arbeitet.

3.2 Hardwarevoraussetzungen

Wie schon erwähnt, bearbeitet QuickDraw einzelne Bits im Hauptspeicher des Macintosh. Den Grafikroutinen ist es dabei völlig egal, was für Bits das sind. Es kann sich um den Bildschirmpuffer, einen Druckerpuffer oder einen beliebigen vom Programmierer reservierten Speicherbereich handeln, in dem "gezeichnet" wird. Man sieht aber natürlich nur dann etwas, wenn es sich dabei "zufällig" um einen Speicherbereich handelt, der auf dem Bildschirm dargestellt wird.

Die einzige "Hardware"-Voraussetzung für QuickDraw ist die freie Verfügbarkeit über einen gewissen Teil des Hauptspeichers. Erst die Video-Schaltungen des Macintosh sorgen dafür, daß eine bestimmte Zuordnung zwischen einem Speicherbereich und dem Bildschirm stattfindet. Diese Zuordnung sieht im Moment so aus, daß Bits, deren Wert 1 ist, auf einen schwarzen und 0-Bits auf einen weißen Punkt am Bildschirm abgebildet werden. QuickDraw "interessiert" diese Zuordnung jedoch nicht! Genausogut könnten 0-Bits auch schwarzen Punkten oder mehrere Bits nur einem Bildschirm-Punkt zugeordnet werden.

Wichtig für QuickDraw ist aber ein anderer Aspekt: Im Speicher des Macintosh sind die Bits ja mehr oder weniger hintereinander angeordnet. Jeweils 8 Bits werden zu einem Byte zusammengefaßt, 2 Byte zu einem Wort usw. Die Bytes liegen bei bestimmten Adressen, die man angeben muß, wenn man ihren Inhalt lesen oder schreiben will. Diese Adressen sind einfach

aufsteigende ganze Zahlen. QuickDraw betrachtet den Speicher jedoch nicht als eine lange, lineare Liste von Bits, sondern als "Tabelle", in der die Bits in Zeilen und Spalten angeordnet sind. 2 Bit liegen für QuickDraw also nicht nur hintereinander und haben einen bestimmten Abstand voneinander. Sie können auch untereinander liegen, und Abstände müssen horizontal und vertikal getrennte Komponenten haben. Sobald eine Folge von Bits im Speicher nicht mehr einfach als hintereinanderliegend aufgefaßt wird, sondern ihr eine rechteckige Anordnung gegeben wird, spricht die QuickDraw-Dokumentation von einem "BitImage" (*image* = engl. für Bild; Abbildung). Wir machen uns dann praktisch in Gedanken bereits eine Vorstellung davon, wie diese Bits auf dem Bildschirm dargestellt werden.

Als einfachstes Beispiel für ein solches BitImage soll uns ein Pattern (engl. für Muster) dienen. Pattern finden auf dem Macintosh Verwendung zum Zeichnen von Linien und Füllen von Flächen etc. Es sind einfache Muster von schwarzen und weißen Punkten, mit denen man Vielfalt und Unterscheidungsmöglichkeiten in eine Zeichnung bringen kann, obwohl der Macintosh keine farbigen Darstellungen kennt. Der Typ Pattern ist definiert wie folgt:

```
TYPE Pattern = PACKED ARRAY [0..7] OF 0..255;
```

Es handelt sich also laut Definition um eine Folge von 8 Bytes (0..255). Die folgenden 8 Byte, die im Hauptspeicher des Macintosh bei der Adresse 15467 beginnen, wären also ein Muster.

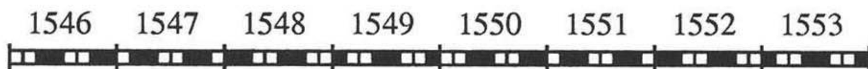


Bild 3 - 4: *Ein Pattern als Bytefolge*

Um welches Muster es sich handelt, wird einem aber erst dann klar, wenn man die Bytes nicht einfach hintereinanderlegt, wie in der obigen Abbildung, sondern die Bytes zu Zeilen anordnet — in diesem Fall jeweils 1 Byte in eine Zeile:

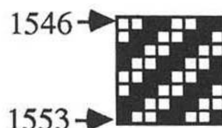


Bild 3 - 5: *Ein Pattern als BitImage*

Die relativ unübersichtliche Folge von Bits bzw. Bytes aus Abbildung 3 - 4 stellte sich also als Streifenmuster heraus. Mit 8 * 8 Bit bzw. schwarzen und weißen Punkten kann man also ein quadratisch angelegtes Muster definieren. Durch Nebeneinanderlegen solcher Quadrate können dann beliebig große Flächen mit einem solchen Muster gefüllt werden. Um es dem Programmierer leichter zu machen, sind die Muster **white** (Weiß) , **ltGray** (Hellgrau), **gray** (Mittelgrau), **dkGray** (Dunkelgrau) und **black** (Schwarz) bereits vordefiniert.

Mit dem Wissen um den Aufbau eines Patterns ist es aber ein leichtes, sich beliebige zusätzliche Muster zu definieren. Man braucht sich das Muster nur auf ein Blatt Rechenpapier malen und die einzelnen Zeilen als Bytes auffassen. Die den Bytes entsprechenden Zahlenwerte schreibt man dann in ein Patternarray und schon ist ein neues Muster fertig. Ähnlich kann man sich einen neuen Maus-Cursor definieren; nur ist dessen BitImage etwas größer.

Genauso kann man jede andere Bitfolge als BitImage betrachten, wenn man ihr nur eine Zeilenlänge gibt. Verschiedene Zeilenlängen ergeben dabei logischerweise unterschiedliche Bilder, und die meisten Bitfolgen sehen wohl nur bei einer ganz bestimmten Zeilenlänge gut aus. Für Pattern und einige andere "primitive" BitImages liegt die Zeilenlänge fest. Bei anderen BitImages, die wir bald kennenlernen werden, kann man QuickDraw die exakte Zeilenlänge mittteilen.

3.3 Zusammenhänge zwischen Geometrie und Bits: BitMaps

Bis jetzt haben wir zwei ganz getrennte Teilgebiete, die von QuickDraw berührt werden, behandelt. Da waren auf der einen Seite die reinen mathematischen Objekte wie Flächen, Rechtecke und Punkte und auf der anderen Seite die einzelnen Bits, die wir zu BitImages zusammenfassen können und die über die Video-Hardware dann zu Bildern am Bildschirm des Macintosh werden.

Was fehlt, ist die Verbindung zwischen diesen Teilgebieten; eine Reihe von Regeln bzw. Angaben, die QuickDraw sagen, welche mathematischen Objekte er welchen Bits in einem BitImage zuordnen soll. Das wichtigste BitImage, das uns dabei interessiert, ist der Bildschirmpuffer des Macintosh, damit wir auch etwas von unseren Zeichnungen sehen. Diese gesuchte Verbindung schaffen die sogenannten BitMaps (engl. für Bit-Karten). Eine BitMap dient dazu, QuickDraw mitzuteilen, welcher Bereich der abstrakten Grafikebene in einem bestimmten BitImage — z.B. dem Bildschirmpuffer — dargestellt wird, und schafft eine Zuordnung von Punkten der Grafikebene zu Positionen am Bildschirm.

```
TYPE BitMap=
       RECORD
          baseAddr:     QDPtr;
          rowBytes:     Integer;
          bounds:       Rect;
       END {OF RECORD BitMap};
```

Das Feld **baseAddr** zeigt auf den Beginn des BitImages dieser Bitmap. Die an dieser Stelle stehenden und darauf folgenden Speicherzellen enthalten die Bits des BitImages. Das Feld **rowBytes** gibt an, nach wie vielen Bytes eine neue Zeile beginnt. Das Feld **bounds** zwingt der Bitmap ein Koordinatensystem auf und bestimmt, wie viele der auf **baseAddr** folgenden Speicherzellen zum BitImage dieser Bitmap gehören.

Man muß sich das so vorstellen, daß sich in der oberen linken Ecke des Rechtecks **bounds** das erste Bit (der schwarze Punkt in Bild 3 - 6), das im Speicher an der Stelle **baseAddr** steht) befindet. Dann folgen **(rowBytes * 8 – 1)** Bits rechts daneben. Das Bit an der Stelle **baseAddr + (rowBytes * 8)** liegt dann genau unter dem ersten Bit usw. usw. Dabei müssen (aus technischen Gründen) sowohl **baseAddr** eine gerade und **rowBytes** eine durch 4 teilbare Zahl sein.

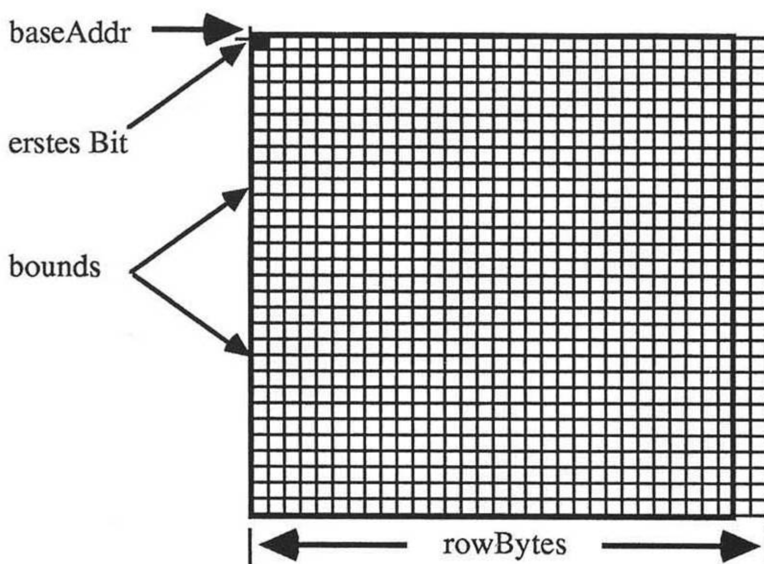


Bild 3 - 6: *Eine Bitmap*

Das in der Abbildung dargestellte Raster ist das Raster der virtuellen Grafikebene. In der Datenstruktur der Bitmap wird die Verbindung zwischen der mathematisch definierten Grafikebene und den Bits im Speicher des Macintosh festgelegt. Zum Beispiel wird durch **bounds** (in der Abbildung durch etwas stärker gezogene Linien hervorgehoben) festgelegt, welches Rechteck in der Grafikebene von den Bits der Bitmap eingenommen wird. Die Bits entsprechen dabei der weißen Fläche zwischen den Rasterlinien. Sie (und damit auch die Punkte auf dem Bildschirm) liegen also nicht auf den Punkten der Grafikebene, sondern dazwischen. Dies ist auch im mathema - tischen Sinne korrekt, da ein Punkt im Sinne der euklidschen Geometrie ja keine Ausdehnung hat, ein Bildschirm-Punkt aber schon. Ein Punkt auf dem Bildschirm ist also kein QuickDraw-Punkt, sondern das kleinstmögliche Rechteck, bei dem beide Seitenlängen gleich 1 sind.

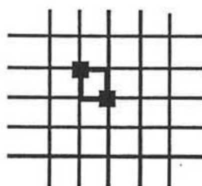


Bild 3 - 7: *Ein Bildschirm-Punkt*

Da es sehr mißverständlich ist, sowohl von Bildschirm-Punkten, Bits und auch von QuickDraw-Punkten zu sprechen, werde ich in Zunkunft nur noch von Punkten sprechen, wenn damit Koordinaten in der Grafikebene, also QuickDraw-Punkte, gemeint sind. Bildschirm-Punkte werde ich generell "Pixel" und gelegentlich auch "Bits" nennen, wenn es um Speicherstrukturen geht.

3.4 Der Aufbau eines GrafPorts

Wann immer mit QuickDraw-Befehlen etwas gezeichnet wird, so geschieht dies immer mit Bezug auf den aktuellen "GrafPort". Es können zu jeder Zeit mehrere GrafPorts existieren, zwischen denen man per Funktions-Aufruf hin und her wechseln kann. Je nachdem in welchem man zeichnet, können dieselben Befehle völlig unterschiedliche Ergebnisse haben.

3.4.1 Überblick über die Felder eines GrafPorts

Der GrafPort stellt zum Beispiel den Bezug zwischen der abstrakten Grafikebene und den Punkten des Bildschirms her. Ein Feld eines GrafPorts enthält nämlich eine Bitmap, die angibt, auf welchen Bits QuickDraw arbeitet, wenn dieser Port aktuell ist.

Die anderen Felder eines GrafPorts enthalten vor allem Angaben über die Bereiche, in denen überhaupt Zeichen-Operationen stattfinden dürfen und eine Reihe von weiteren (vor allem Zustands-) Informationen, die den Umgang mit QuickDraw etwas bequemer machen. Die genaue Struktur beschreibt der folgende Pascal-Record:

```
TYPE GrafPtr =  ^GrafPort;
     GrafPort =
```

```
RECORD
    device:      INTEGER;
    portBits:    BitMap;
    portRect:    Rect;
    visRgn:      RgnHandle;
    clipRgn:     RgnHandle;
    bkPat:       Pattern;
    fillPat:     Pattern;
    pnLoc:       Point;
    pnSize:      Point;
    pnPat:       Pattern;
    pnVis:       INTEGER;
    txFont:      INTEGER;
    txFace:      Style;
    txMode:      INTEGER;
    txSize:      INTEGER;
    spExtra:     INTEGER;
    fgColor:     LONGINT;
    fgColor:     LONGINT;
    colrBit:     INTEGER;
    patStretch:  INTEGER;
    picSave:     QDHandle;
    rgnSave:     QDHandle;
    polySave:    QDHandle;
    grafProcs:   QDProcsPtr;
  END {OF RECORD GrafPort};
```

Ich möchte die wichtigsten Felder eines GrafPorts nun kurz erläutern. Einige werden nur kurz angeschnitten, da sie weiter unten im Zusammenhang mit den QuickDraw-Prozeduren noch näher erklärt werden.

Das Feld **device** gibt an, auf welchem physikalischen Medium die BitMap dargestellt wird, und ist nur für den internen Gebrauch gedacht.

3.4.2 Begrenzungen der Zeichenroutinen

Die Felder **portBits, portRect, visRgn** und **clipRgn** dienen hauptsäch - lich dazu, den genauen Bereich festzulegen, in dem QuickDraw arbeitet, wenn dieser GrafPort aktuell ist.

portBits ist eine BitMap, die einen Teil der QuickDraw-Grafikebene auf einen zusammenhängenden Speicherbereich im Macintosh abbildet. **portBits** zeigt zumeist auf den Bildschirmpuffer und ist nur für fortgeschrittene Anwendungen interessant. Wichtig aber ist, daß das Rechteck **portBits.bounds** ein Koordinatensystem für diesen GrafPort vereinbart. Das erste Pixel in der BitMap liegt direkt rechts neben und unterhalb des Punktes **portBits.bounds.topLeft**.

portRect ist ein Rechteck, das einen Teil der BitMap für die Benutzung durch diesen GrafPort "freigibt". Die Koordinaten von **portRect** müssen in dem System betrachtet werden, wie es von **portBits.bounds** definiert wird. **portRect** liegt sinnvollerweise innerhalb von **portBits.bounds**; dies muß aber nicht notwendigerweise so sein. Nur die Bits der BitMap, die innerhalb von **portRect** liegen, dürfen durch QuickDraw-Operationen verändert werden.

visRgn gibt den Bereich der BitMap an, der von der BitMap dieses GrafPort überhaupt auf dem Bildschirm sichtbar ist, und dient ebenfalls dazu, Zeichnungen einzuschränken. Die QuickDraw-Routinen verwenden dieses Feld, um nicht überflüssigerweise in Bereichen zu arbeiten, die sowieso nicht sichtbar sind — z.B. weil sie von einem anderen Fenster "überdeckt" werden.

Und mit **clipRgn** kann ein Anwendungsprogramm den "zum Zeichnen freigegebenen" Teil der BitMap noch weiter einschränken. Wenn man z.B. eine Scheibe aus einem Kreis malen möchte, so kann man dies sehr einfach tun, indem man zunächst ein Rechteck als clipRgn definiert, innerhalb dessen der Teil des Kreises liegt, der sichtbar sein soll, und dann einen ganzen Kreis zeichnet.

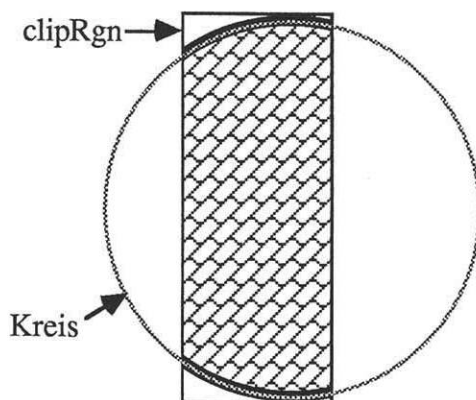


Bild 3 - 8: *Die Wirkung der clipRgn*

Zusammenfassend kann man sagen, daß QuickDraw nur innerhalb der Schnittfläche von **portBits.bounds, portRect, visRgn** und **clipRgn** überhaupt etwas tut. Zeichnet man z.B. eine Linie und sie verläßt diese Schnittfläche, so hört QuickDraw einfach auf zu zeichnen. Um zu verstehen, warum diese Einschränkungsmöglichkeiten wichtig sind und so viele verschiedene Felder gebraucht werden, muß ich allerdings etwas vorgreifen und Fenster ins Spiel bringen. Die Hauptverwendung von GrafPorts geschieht nämlich auf dem Macintosh in Fenstern. Ein Fenster ist im wesentlichen ein GrafPort.

Bei Fenstern ist **portBits** stets der gesamte Bildschirmpuffer, und **bounds** ist das Rechteck ((0,0),(512,342)). Das Rechteck, das im Innern des Fensters gerade dargestellt ist, wird durch **portRect** bestimmt. Wird dieses Fenster nun durch ein oder mehrere andere Fenster teilweise verdeckt, so enthält **visRgn** den Teil von **portRect**, der nicht verdeckt ist. **clipRgn** schließlich ist zur freien Verwendung für den Programmierer gedacht und wird von der Fensterverwaltung nicht gebraucht. Die folgende Abbildung versucht, diese recht komplizierten Zusammenhänge in grafischer Form darzustellen.

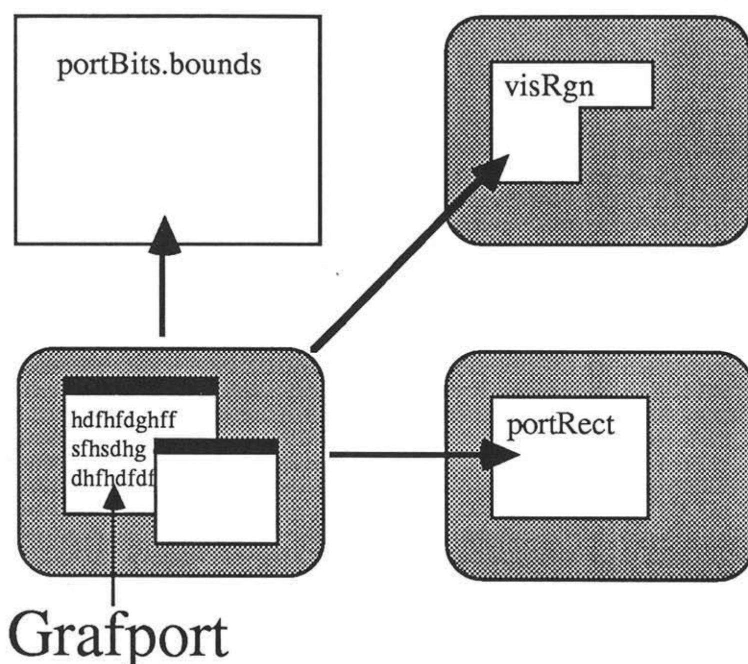


Bild 3 - 9: *Aufbau der wichtigsten GrafPort-Regionen*

Links unten ist das Bild zu sehen, wie es dem Betrachter auf dem Bildschirm gezeigt wird. Darum herum sind die daraus folgenden Regionen für den GrafPort des verdeckten Fensters angedeutet. Möchte der Programmierer beim Zeichnen zusätzlich noch einen weiteren Teil des Fensters abdecken, so kann er dazu die **clipRgn** verwenden. Diese Möglichkeit wird allerdings nur in seltenen Ausnahmefällen gebraucht.

Die Begrenzungen des GrafPorts sind für den Programmierer nur dann wichtig, wenn er das Koordinatensystem innerhalb des Fensters verändern will (das von **bounds** bestimmt wird) oder feststellen möchte, wie groß das aktuelle Fenster wirklich ist. Bis auf **clipRgn** werden alle diese Rechtecke und Regionen ansonsten nur von der Fensterverwaltung gepflegt und vom Programm aus nicht geändert.

Alle Zeichnungen erfolgen üblicherweise nämlich ohne Rücksicht auf irgendwelche Beschränkungen. Der Programmierer geht davon aus, daß er zum Zeichnen die ganze virtuelle Grafikebene zur Verfügung hat. Welcher Ausschnitt davon gerade im Fenster gezeigt werden soll (**portRect**) oder

welcher Teil des Fensters sichtbar ist (**visRgn**), ist dabei nicht so wichtig. QuickDraw sorgt von sich aus dafür, daß alle Ausgaben nur auf den sichtbaren Teil des Fensters beschränkt bleiben und keine Grafik außerhalb dieser Region stören. Aus Effizienz-Gründen (Zeit) kann es aber sinnvoll sein, nachzuprüfen, ob überhaupt ein Teil einer komplizierten Grafik sichtbar ist, bevor man beginnt, ihn zu zeichnen.

3.4.3 Der Zeichenstift

In den folgenden Feldern eines GrafPorts wird der Zustand eines gedachten Zeichenstiftes, des sog. "Pens", gemerkt. Dieser Stift funktioniert in etwa wie der Zeichenstift von MacPaint. Er kann bewegt werden, ohne daß gezeichnet wird (wie in MacPaint mit gelöster Maustaste), und er kann bewegt werden und zeichnet dabei eine Linie von bestimmter Höhe und Breite mit dem aktuellen Muster (wie in MacPaint mit gedrückter Maustaste). Hauptunterschiede zum Stift in MacPaint liegen nur darin, daß der QuickDraw-Pen nicht mit der Maus, sondern durch Funktionsaufrufe bewegt wird und nicht so viele verschiedenen Formen annehmen kann; er ist stets rechteckig.

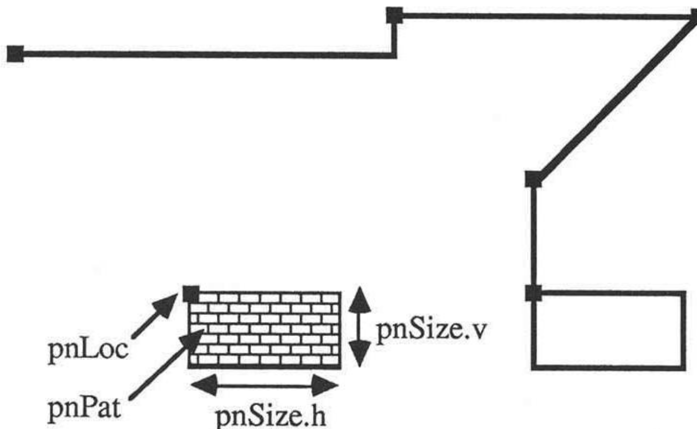


Bild 3 - 10: *Der QuickDraw-Zeichenstift*

In **pnLoc** steht die aktuelle Position der oberen, linken Ecke dieses Stiftes und in **pnSize** seine Größe. In **pnPat** steht das aktuelle Muster (Pattern), mit dem gezeichnet wird, und in **pnVis**, ob der Stift wirklich zeichnet, wenn er bewegt wird. Ist **pnVis** >= 0, so wird gezeichnet. Ist **pnVis** < 0, so

bleibt der Strich unsichtbar. Der Pen ist praktisch ein Rechteck, dessen linke, obere Ecke stets bei **pnLoc** steht. Wird dieses Rechteck bewegt, so wirkt der Stift immer auf die Bits, die er überstreicht — die also im Laufe seines Weges irgendwann einmal innerhalb des Rechtecks liegen. Wie aus Bild 3 - 9 deutlich zu sehen ist, zeichnet er deshalb immer rechts und unterhalb vom Punkt **pnLoc** .

Wie der Stift genau auf die überstrichenen Pixel in der Bitmap wirkt, wird durch die Felder **pnPat** und **pnMode** sowie durch die schon vorhandenen Pixel festgelegt. QuickDraw legt zunächst das Pen-Rechteck über die Bitmap und nimmt sich jeweils ein Pixel aus der Bitmap und das drüberliegende Pixel aus dem mit pnPat gefüllten Rechteck. Diese beiden Pixel werden logisch verknüpft, und das alte Pixel in der Bitmap wird gegen das Ergebnis dieser Verknüpfung ersetzt. Welche Verknüpfung zwischen den beiden Pixeln durchgeführt wird, hängt vom **pnMode** ab. Es gibt dafür insgesamt 8 Möglichkeiten:

pnMode:	Farbe des Pixels in der Bitmap, wenn Pattern-Pixel schwarz	Pattern-Pixel weiß
patCopy	schwarz	weiß
patOr	schwarz	unverändert
patXor	invertiert	unverändert
patBic	weiß	unverändert
notPatCopy	weiß	schwarz
notPatOr	unverändert	schwarz
notPatXor	unverändert	invertiert
notPatBic	unverändert	weiß

Die mit **not...** beginnenden Modi tun im wesentlichen dasselbe wie ihre "normalen" Varianten, sie invertieren nur zuvor das Pixel im **pnPat**. Auf die drei wichtigsten der normalen Modi will ich noch einmal kurz eingehen:

• Im Modus **patCopy** wird einfach jedes Pixel aus der Bitmap gegen das entsprechende Pixel aus dem **pnPat** ersetzt. Anders, als man es vielleicht erwartet, ersetzen auch weiße Pixel die schwarzen Pixel in der Bitmap.

• Der Modus **patOr** wirkt wie normales Zeichnen mit dem Bleistift auf Papier. Schwarze Pixel aus dem **pnPat** ersetzen die durunterliegenden Pixel aus der Bitmap, weiße ändern nichts.

• Der Modus **patXor** wird vor allem dazu gebraucht, ein vorhandenes Bild ändern zu können und diese Änderung danach leicht wieder rückgängig machen zu können. Zieht man einen schwarzen Strich mit **pnMode** = **patXor** , so wird jedes Pixel in der Bitmap "umgeklappt" oder "invertiert", weiße Pixel werden schwarz, schwarze werden weiß. Zieht man gleich danach denselben Strich noch einmal mit **patXor**, so ist wieder alles beim alten.

Diese ganzen Statusinformationen werden hauptächlich nur zur Bequemlich - keit des Programmierers in Feldern des GrafPorts gespeichert. Würde QuickDraw sie nämlich nicht dort merken, müßten sie bei jedem Funktions-Aufruf, der diese Informationen braucht, extra mitangegeben werden.

3.4.4 Der Text-Stift

Unabhängig vom Zeichenstift gibt es noch einen separaten Text-Stift, dessen Zustand ebenfalls im GrafPort gespeichert wird. Während der normale Zeichenstift beim Zeichnen von Linien, Kreisen, Rechtecken etc. Verwendung findet, gelten für die Ausgabe von Texten eigene Regeln. Die aktuelle Position des Zeichenstiftes wird allerdings von Textausgaben und anderen Grafikroutinen geteilt. Jede Textausgabe beginnt bei **pnLoc** und verändert diese Position auch.

Um die Textausgabe auf dem Macintosh zu verstehen, muß man zunächst einmal sehen, daß Buchstaben und andere Zeichen von QuickDraw genauso behandelt werden wie alle anderen Grafiken auch. Es gibt keinen bestimmten Modus, indem auf dem Bildschirm nur Text erscheint, und Buchstaben müssen auch nicht in ein bestimmtes Raster (z.B. 8 x 8 Punkte) passen. Jedes Zeichen ist ein kleines Bild (ein BitImage), das bei der Textausgabe direkt auf den Bildschirm kopiert wird. Ein komplettes Alphabet (incl. aller Sonderzeichen) mit solchen Bildern wird jeweils in einem Zeichensatz zusammengefaßt. Ein Zeichensatz ist somit eine Sammlung von BitImages der verschiedenen Buchstaben, Zahlen und Sonderzeichen. Die folgende Abbildung zeigt die wesentlichen Eigenschaften eines Zeichens bzw. eines Zeichensatzes.

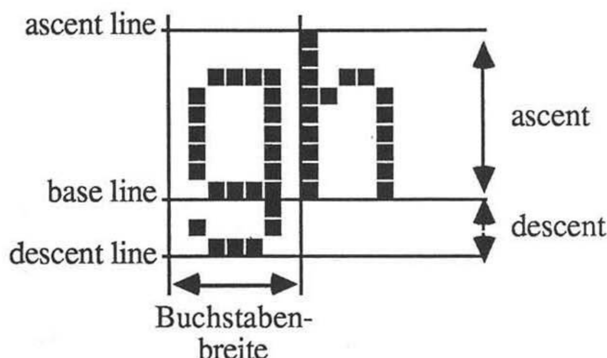


Bild 3 - 11: *Aufbau eines Buchstabens*

Die **base line** bildet praktisch die Null-Linie, von der die Zeichen aus gezeichnet werden. **pnPos** liegt immer auf der **base line**. Die größten Zeichen eines Zeichensatzes erstrecken sich maximal bis zur **ascent line**, und Unterlängen wie beim Buchstaben "g" können bis zur **descent line** hinuntergehen. Diese Unterscheidung zwischen oberem und unterem Teil eines Zeichens ist z.B. für das Unterstreichen und Kursiv-Schreiben wichtig. Jedes Zeichen hat zudem eine bestimmte Breite, die nicht für alle Zeichen gleich sein muß (proportionale Zeichensätze sind möglich).

Nun aber zu den Feldern des GrafPorts, die für die Darstellung von Texten auf dem Bildschirm zuständig sind:

• Das Feld **txFont** gibt an, mit welchem Zeichensatz (engl. "Font") die nächste Textausgabe erfolgen soll. Dieses Feld enthält eine ganze Zahl (Typ INTEGER) und keinen Zeichensatz-Namen, wie sie z.B. im Zeichensatz-Menü von MacWrite auftauchen. Welche Zahl welchem Zeichensatz entspricht, kann man für die "normalen" Zeichensätze, die auf der System-Diskette mitgeliefert werden, im Handbuch von QuickDraw nachschlagen. Da Apple aber nicht alle Zeichensätze erahnen kann, die es irgendwann einmal für den Macintosh geben wird, kann man die zu einem bestimmten Zeichensatz-Namen gehörige Nummer auch über eine spezielle Funktion erfahren. (Es handelt sich dabei um die Funktion **GetFNum**, die in *Inside Macintosh* im Kapitel *FontManager* dokumentiert ist. Da, außer bei sehr speziellen Anwendungen, nur diese eine Funktion des FontManagers gebraucht wird, möchte ich ansonsten nicht weiter auf seine Aufgaben eingehen.)

Text kann auf dem Macintosh in einer ganzen Reihe von verschiedenen Arten ausgegeben werden. Hierfür ist das Feld **txFace** zuständig. **txFace** ist vom Typ **Style**, der wie folgt definiert ist:

```
TYPE StyleItem =  (bold,italic,underline,outline,
                   shadow,condense,extend );
     Style     =    SET OF StyleItem;
```

Style ist also eine Menge von (SET OF) Eigenschaften, die das Aussehen eines Textes beeinflussen können. Jeder auf dem Bildschirm ausgegebene Text kann eine beliebige Kombination dieser Eigenschaften besitzen.

Normale Schrift (txFace = [ ])
Fette Schrift (txFace = [bold])
Unterstrichene Schrift (txFace = [underline])
Kursive Schrift (txFace = [italic])
Outlined Schrift (txFace = [ouline])
Schatten-Schrift (txFace = [shadow])
Fette, unterstrichene Kursiv-Schrift
(txFace = [bold, italic, underline])

Bild 3 - 12: *Die verschiedenen Stil-Variationen für Textausgaben*

Gibt man **bold** an, so wird jeder ausgegebene Buchstabe mehrmals geschrieben und dazwischen jedesmal um genau ein Pixel nach rechts verschoben. Dies läßt ihn kräftiger bzw. dunkler erscheinen. Wie viele Pixel die Buchstaben dicker werden, ist von Zeichensatz zu Zeichensatz verschieden.

Bei Textausgaben in **italic** werden alle Buchstaben in kursiv gedruckt. Hierzu werden alle Pixel unterhalb der base line etwas nach links und alle darüber etwas nach rechts verschoben. Die Verschiebung ist um so größer, je größer die Entfernung der Pixel von der *base line* ist. Wie stark genau die Schräglage ist, hängt ebenfalls vom Font ab. Anzumerken bleibt zu **italic** vor allem noch, daß es meistens weder auf dem Bildschirm noch auf dem Matrixdrucker besonders gut aussieht und oft auch schwer leserlich wird. Dies liegt ganz klar an der Hardware. Leserliche und gut aussehende Kursiv-Schrift ist erst bei einer höheren Auflösung (kleineren Punkten) des

Ausgabemediums zu erwarten — etwa beim LaserWriter, mit dem dieses Buch erstellt wurde.

Beim Stil **underline** wird ein zusätzlicher Strich unter jedem Buchstaben gezogen. Zwischen der base line und dem Strich bleibt normalerweise ein Pixel frei, und der Strich ist auch ein Pixel dick. Dies hängt aber auch vom verwendeten Zeichensatz ab. Der Unterstrich geht übrigens nicht durch Unterlängen hindurch, sondern läßt links und rechts neben den Unterlängen noch 1 Pixel frei.

Die Variationen **outline** und **shadow** werden wohl selten für normale Textausgaben gebraucht werden. Sie eignen sich aber für interessante Überschriften. **outline** zeichnet um jeden Buchstaben einen schwarzen Rand und setzt den Buchstaben dann selbst in Weiß in diesen Rahmen hinein. **shadow** arbeitet im Prinzip genauso, nur ist der schwarze Rand unten und rechts etwas dicker, um einen räumlichen Eindruck (Schatten) hervorzurufen.

Die Eigenschaften **condensed** und **extended** schließlich sorgen dafür, daß hintereinandergeschriebene Buchstaben dichter aneinander oder weiter auseinander geschrieben werden. Diese Möglichkeiten werden nur selten gebraucht, da sie meist nicht besonders leserlich sind, und werden z.B. von den meisten Textverarbeitungsprogrammen nicht unterstützt.

• Das Feld **txMode** arbeitet genauso wird **pnMode,** nur eben für Textausgaben und nicht für Linien oder Kreise. Schreibt man Text z.B. im Modus **patCopy** auf den Bildschirm, so wird aller schon an der selben Stelle stehende Text gelöscht. Schreibt man hingegen mit **patOr**, so wird der neue Text über den alten geschrieben, genau wie man mit der Schreibmaschine zwei Buchstaben übereinanderschreiben kann, wenn man den Wagen wieder zurückbewegt.

• **txSize** gibt an, wie groß die Buchstaben sein sollen. Die Einheit, in der dies angegeben wird, ist der typografische "Punkt", was ungefähr einem 1/72 Inch oder einem Pixel auf dem Bildschirm entspricht. Die üblichen Größen sind dabei 9, 10, 12, 14, 18, 20, 24, 36, 48 und 72 Punkte. Gibt man eine Größe, die nicht in dieser Aufzählung enthalten ist, an oder verwendet man eine Größe, die nicht als Zeichensatz-Größe in der Systemdatei verzeichnet ist, so bekommt QuickDraw Probleme.

Die BitImages für die einzelnen Buchstaben sind dann nämlich nicht vorhanden, sondern nur für größere oder kleinere Buchstaben. Statt dessen wird dann eine vorhandene Größe verwendet, und die entsprechenden Buchstaben werden gedehnt oder gestaucht. Der Vorgang ist ziemlich

derselbe wie beim Verzerren eines Bild-Bereichs in MacPaint. Die Ergebnisse dieser Verzerrung sehen — wie aus MacPaint bekannt — nicht immer sehr gut aus. Es ist deshalb ratsam, nur wirkliche vorhandene Zeichensatz-Größen zu nehmen. Einigermaßen akzeptabel sehen die Ergebnisse allerdings aus, wenn ein Zeichensatz in genau der doppelten oder halben Größe vorhanden ist. Nach diesen sucht QuickDraw deshalb auch zunächst, wenn es die gewünschte Größe nicht findet.

• Das letzte Feld des GrafPorts, das etwas mit Textausgabe zu tun hat, ist **spExtra**. Der Inhalt dieses Feldes bestimmt, um wieviel Pixel durch ein Leerzeichen (engl. **space**) getrennte Worte zusätzlich weiter auseinander - gezogen werden, wenn ein Text aus mehreren Worten ausgeben wird. Dies dient vor allem dazu, um auf einfache Weise Blocksatz in der Textverarbeitung zu erzielen. **SpExtra** darf nicht mit dem Stil **extend** verwechselt werden, der alle Buchstaben weiter auseinanderzieht. Um die Angabe, um wieviel die trennenden Leerzeichen breiter werden sollen, möglichst genau zu ermöglichen, wird **spExtra** von QuickDraw als Fixpunkt-Zahl interpretiert. (Fixpunkt-Zahlen werden im Abschnitt *ToolBox-Utilities* von *Inside Macintosh* behandelt. Sie werden zu selten gebraucht, als daß ihre ausführliche Erörterung in dieser Einführung gerechtfertigt wäre.)

3.5 Elementare Zeichenoperationen

QuickDraw kennt im wesentlichen die folgenden Elemente (Objekte), aus denen sich eine Grafik zusammensetzen kann:

- Linien
- Rechtecke
- abgerundete Rechtecke
- Ovale
- Oval-Ausschnitte
- Polygone
- Regionen
- komplette Bilder

  ...... und natürlich Texte.

Mit jedem dieser Grundelemente (außer Texten und Bildern) sind die folgenden Operationen möglich:

- einen Umriß des Elements zeichnen
- die umschlossene Fläche mit einem Muster füllen

- die umschlossene Fläche löschen
- die umschlossene Fläche invertieren

Jede Art von Objekt kann mit jeder Operation beliebig verknüpft werden. Ich möchte im folgenden die einzelnen Operationen mit jeder der möglichen Objekt-Arten vorstellen. Im Abschnitt über Rechtecke werde ich etwas ausführlicher auf die Operationen eingehen, danach im wesentlichen nur noch den Aufbau der Objekte beschreiben, da die Wirkung der Operationen doch immer recht ähnlich ist.

3.5.1 Linien und Bewegungen

Mit die einfachsten QuickDraw-Funktionen sind diejenigen, die den Zeichenstift bewegen. Der Zeichenstift wird mit diesen Funktionen stets von einem Punkt zu einem anderen bewegt, d.h. der Wert des GrafPort-Feldes **pnLoc** wird geändert. Die mit **Move...** beginnenden Operationen ändern praktisch nur dieses Feld, während die mit **Line...** beginnenden Funktionen dabei eine Linie zwischen dem Startpunkt und dem Zielpunkt zeichnen. Das Aussehen der Linie wird dabei von der Feldern **pnPat, pnSize**, und **pnMode** bestimmt, wie im Abschnitt 3.4.3 über die Eigenschaften des Zeichenstiftes bereits erläutert wurde. Alle Bewegungen können sowohl relativ zur aktuellen Position (**pnLoc**) erfolgen, als auch absolut, d.h. hin zu einem bestimmten Punkt der Grafikebene, dessen Koordinaten bekannt sind.

`PROCEDURE MoveTo (h: INTEGER; v: INTEGER);`

Hierdurch wird die aktuelle Position des Zeichenstiftes auf den Punkt (h,v) gesetzt. Es handelt sich um eine reine Bewegungsoperation. Auf keinen Fall wird dadurch irgend etwas gezeichnet.

`PROCEDURE Move (dH: INTEGER; dV: INTEGER);`

Move verändert die aktuelle Position des Zeichenstiftes um die angegebenen Beträge. Es entspricht der Operation **MoveTo (pnLoc.h + dH, pnLoc.v + dV).**

`PROCEDURE LineTo (h: INTEGER; v: INTEGER);`

LineTo zieht eine Linie von der aktuellen Position des Zeichenstiftes zum angegebenen Punkt. Höhe, Breite und das Muster der Linie werden von den Werten der Felder **pnSize** und **pnPat** des GrafPorts bestimmt und die Wirkung auf die schon vorhandenen Pixel auf der Linie durch den Wert des Feldes **pnMode**. Nach Ausführung der Operation ist **pnLoc** = (h,v).

`PROCEDURE Move (dH: INTEGER; dV: INTEGER);`

Line arbeitet ähnlich wie **Move** relativ zur aktuellen Position. Es entspricht der Operation **LineTo (pnLoc.h + dH, pnLoc.v + dV).**

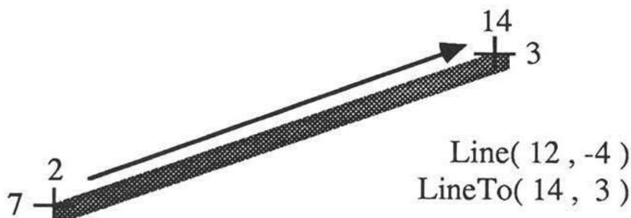


Bild 3 - 13: *Das Zeichnen einer Linie mit QuickDraw*

Wie in der Abbildung hoffentlich deutlich zu sehen ist, werden von einem **Line...** Befehl nur diejenigen Punkte betroffen, die während der Bewegung des Stiftes irgendwann einmal in dem Rechteck zu liegen kommen, das der Stift auf der Grafikebene gerade einnimmt. Dieses Rechteck wird dabei bestimmt durch den Wert der Felder **pnLoc**, der die obere, linke Ecke dieses Rechtecks angibt, **pnSize.h,** der die Breite dieses Rechtecks bestimmt, und **pnSize.v**, der die Höhe angibt.

```
PROCEDURE PenSize   (h: INTEGER; v: INTEGER);
PROCEDURE PenPat    (pat: Pattern);
PROCEDURE PenMode   (mode: INTEGER);
```

Diese drei Prozeduren dienen dazu, die Eigenschaften des Stiftes durch einen kurzen Prozeduraufruf zu verändern. Man könnte dies allerdings genauso durch eine direkte Modifikation der entsprechenden Felder des aktuellen GrafPorts erreichen.

3.5.2 Rechtecke

Rechtecke sind uns ja schon aus dem vorangegangenen Abschnitt über mathematische Grundlagen von QuickDraw bekannt. Hier soll es nicht so sehr über abstrakte Mathematik gehen, sondern viel mehr um die konkreten Möglichkeiten, mit Rechtecken (Teile von) Grafiken auf den Bildschirm des Macintosh zu bekommen.

Ein Rechteck ist bestimmt durch zwei Punkte oder vier Koordinaten, die in einer Datenstruktur vom Typ **Rect** gespeichert werden können. Das genaue Aussehen dieser Datenstruktur ist ja bereits aus einem früheren Abschnitt bekannt. Zum Rechteck gehört die gesamte Fläche zwischen diesen zwei Punkten. Alle Bits (Pixel) der BitMap, die in dieser Fläche liegen, können von Grafik-Operationen mit Rechtecken beeinflußt werden.

```
PROCEDURE FillRect (r: Rect; pat: Pattern);
```

Die Funktion **FillRect** füllt den gesamten Teil der BitMap, der von diesem Rechteck umschlossen wird, mit dem angegebenen Muster. Die Bits des Musters ersetzen alle Bits, die sich vorher an dieser Stelle befunden haben (im Modus **patCopy**).

```
PROCEDURE EraseRect (r: Rect);
```

Die Funktion **EraseRect** füllt ebenfalls den vom Rechteck umschlossenen Teil der BitMap mit einem Muster (im Modus **patCopy**). Als Muster wird von dieser Funktion das Muster verwendet, das im Feld **bkPat** des aktuellen GrafPorts verzeichnet ist.

```
PROCEDURE PaintRect (r: Rect);
```

PaintRect füllt das Rechteck mit dem im Feld **pnPat** des aktuellen GrafPorts verzeichneten Muster. Beim Füllen wird der in **pnMode** gespeicherte Modus verwendet. Anders als bei **FillRect** werden also die ursprünglich in der BitMap vorhandenen Bits nicht generell ersetzt. Je nach **pnMode** werden sie mit den Bits im Muster **pnPat** logisch verknüpft und erst dann gegen das Ergebnis dieser Verknüpfung ersetzt.

```
PROCEDURE InvertRect (r: Rect);
```

InvertRect invertiert alle vom Rechteck **r** umschlossenen Bits der BitMap; weiße werden schwarz, schwarze weiß.

```
PROCEDURE FrameRect (r: Rect);
```

FrameRect (*Frame* = engl. für Rahmen) zeichnet mit dem aktuellen Stift eine Umriß-Linie um das Rechteck. Der Stift "zieht" dabei an der **Innenseite** des Rechtecks entlang. Die Felder **pnSize**, **pnPat** und **pnMode** bestimmen das Aussehen dieses Rahmens. Wichtig ist die Tatsache, daß alle vom Stift dabei überstrichenen Pixel innerhalb des Rechtecks liegen. Keine Operation mit Rechtecken beeinflußt Pixel außerhab des Rechtecks selbst. Wir erhalten deshalb mit **FrameRect** einen anderen Rahmen, als würden wir nacheinander Linien zwischen den Eckpunkten des Rechtecks ziehen. In diesem Fall würden ja eventuell, je nach Stiftbreite, -muster und -modus, einige Pixel rechts und unterhalb des Rechtecks mit übermalt werden.



Bild 3 - 14: *Die Wirkung von FrameRect*

3.5.3 Abgerundete Rechtecke

Abgerundete Rechtecke (in *Inside Macintosh* werden sie **RoundRects** genannt) sind Rechtecke, deren Ecken durch Ellipsenviertel ersetzt werden. Um ein RoundRect genau zu bestimmen, benötigt man also ein Rechteck und die Angabe des horizontalen und des vertikalen Durchmessers der Ovale.

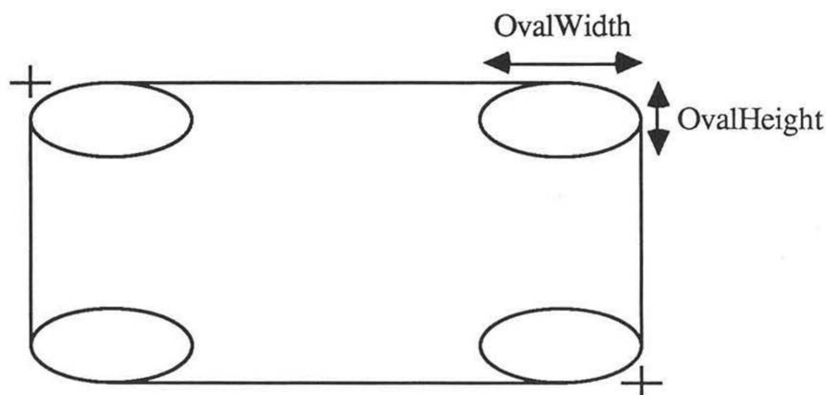


Bild 3 - 15: *Spezifikation eines abgerundeten Rechtecks (RoundRect)*

Die mit RoundRects möglichen Operationen sind ansonsten dieselben wie mit Rechtecken. Es müssen nur immer zwei zusätzliche Parameter mit übergeben werden, die die Ellipsen-Abmessungen angeben.

```
PROCEDURE FillRoundRect
            (r:               Rect;
             ovalWd,ovalHt:  INTEGER;
             pat:             Pattern);
PROCEDURE PaintRoundRect
            (r:               Rect;
             ovalWd,ovalHt:  INTEGER);
PROCEDURE EraseRoundRect
            (r:               Rect;
             ovalWd,ovalHt:  INTEGER);
PROCEDURE InvertRoundRect
            (r:              Rect;
             ovalWd,ovalHt:  INTEGER);
PROCEDURE FrameRoundRect
            (r:              Rect;
             ovalWd,ovalHt:  INTEGER);
```



Bild 3 - 16: *Die Wirkung von FrameRoundRect*

Wichtig ist auch bei RoundRects wieder, daß nur Pixel innerhalb des abgerundeten Rechtecks von irgendeiner Operation betroffen werden können.

3.5.4 Ellipsen und Kreise

Ellipsen und Kreise werden von QuickDraw vollkommen gleich behandelt; der Kreis ist eben ein Sonderfall einer Ellipse — genau wie das Quadrat ein Sonderfall eines Rechtecks ist. In der QuickDraw-Dokumentation heißen sie **Oval**, deshalb will auch ich im folgenden von Ovalen schreiben. Ovale können auch nicht alle Orientierungen in der Ebene annehmen, wie es geometrische Ellipsen könnten. Die beiden Achsen von Ovalen müssen nämlich immer waagerecht oder senkrecht liegen.

Mit dieser Einschränkung wird ein Oval vollständig durch das kleinste Rechteck beschrieben, das es noch vollständig umschließt. In der Mathematik werden Ellipsen zwar üblicherweise anders konstruiert (durch Angabe des Mittelpunkts und zweier Radien oder durch Angabe der zwei Brennpunkte), die Konstruktion mit einem Rechteck hat aber ihre Vorteile. Man kann z.B. durch Angabe eines Rechtecks viel besser abschätzen, wie das zugehörige Oval aussieht, als durch Angabe der beiden Brennpunkte.

Die mit Ovalen möglichen Operationen sind im wesentlichen wieder gleich den Operationen mit Rechtecken:

```
PROCEDURE FillOval    (r: Rect;
                       pat: Pattern);
PROCEDURE PaintOval   (r: Rect);
PROCEDURE EraseOval   (r: Rect);
PROCEDURE InvertOval  (r: Rect);
PROCEDURE FrameOval   (r: Rect);
```



Bild 3 - 17: Die Wirkung von FrameOval

3.5.5 Bögen

Ein Bogen (auf englisch **arc**) ist ein Tortenstück-förmiger Ausschnitt (Sektor) eines Ovals (Ellipse), der durch einen Startwinkel (engl. **startAngle**) und den Bogenwinkel (engl. **arcAngle**) bestimmt ist. Das Oval wird, wie in QuickDraw üblich, durch ein Rechteck angegeben; die beiden Winkel in Grad des Vollkreises von 360°. Der Nullpunkt dieses Kreises liegt bei 12 Uhr, wenn man sich ihn als Ziffernblatt vorstellt, und die Winkel wachsen im Uhrzeigersinn. Eine negative Winkelangabe für **arcAngle** bedeutet deshalb, daß der Bogen, ausgehend von **startAngle**, gegen den Uhrzeigersinn geschlagen werden soll. Die Möglichkeiten zum Zeichnen von Bögen sind ansonsten dieselben wie mit anderen Objekten auch:

```
PROCEDURE FillArc
            (r:             Rect;
             startAngle,
             arcAngle:      INTEGER;
             pat: Pattern);
PROCEDURE PaintArc
            (r:             Rect;
            startAngle,
            arcAngle:       INTEGER);
PROCEDURE EraseArc
            (r:             Rect;
            startAngle,
            arcAngle:       INTEGER);
PROCEDURE InvertArcl
            (r:             Rect;
            startAngle,
            arcAngle:       INTEGER);
PROCEDURE FrameArc
            (r:             Rect;
            startAngle,
            arcAngle:       INTEGER);
```

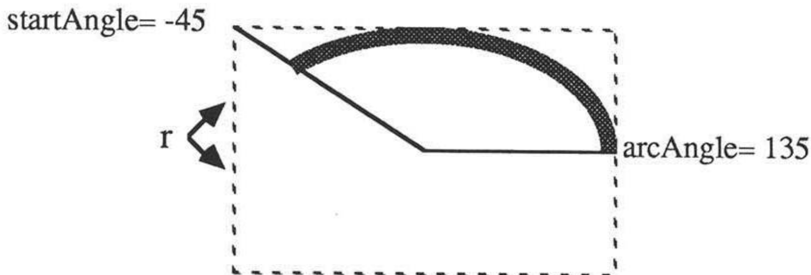


Bild 3 - 18: *Die Wirkung von FrameArc*

Eine Besonderheit von QuickDraw ist allerdings, wie es die Winkelangaben in Abhängigkeit vom angegebenen Rechteck behandelt. Nur wenn das Rechteck ein Quadrat ist, entspricht der Winkel in der Zeichnung unserer Erwartung. Die folgende Abbildung zeigt die Wirkung eines **FillArc** - Befehls, der einen Bogen von mehr als 90° schlagen soll. Betrachten wir den

Winkel in der Zeichnung, so sehen wir aber, daß er deutlich spitzer ist als ein rechter Winkel.

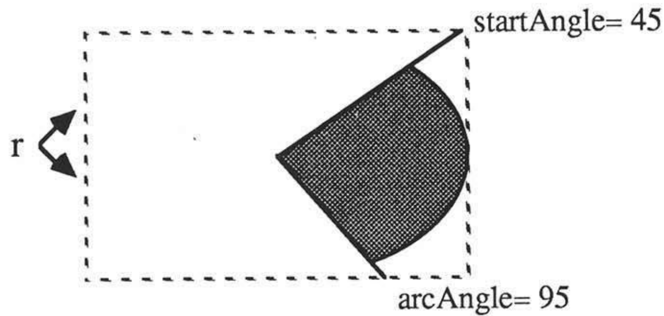


Bild 3 - 19: *Bogenangaben im Rechteck*

Ein **startAngle** von 45° geht z.B. auch immer genau durch die obere, rechte Ecke des Rechtecks, egal wie die Proportionen der Kantenlängen sind. Bei einem sehr flachen Rechteck sehen wir am Bildschirm deshalb einen viel stumpferen Winkel zwischen 0° und 45° als zwischen 45° und 90°.

Die einfachste Vorstellung, diese etwas merkwürdige — aber nichtsdestotrotz manchmal ganz praktische — Auffassung von Winkeln zu verstehen, ist die folgende: QuickDraw zeichnet bei allen Bogen-Funktionen das Bild zunächst in ein Quadrat, dessen Seitenlänge gleich der Breite des für das Oval angegebenen Rechtecks ist. Danach wird der Inhalt dieses Quadrats vertikal so lange gestaucht bzw. gedehnt, bis es genau im Rechteck liegt. Die Stauchung des Quadrats könnte man z.B. auch so auffassen, daß es im Raum gekippt wird, so daß wir nicht mehr ganz senkrecht darauf schauen.

Der Vorteil dieser Auffassung von Winkeln liegt auf der Hand, spätestens, sobald man mit QuickDraw 3D-Grafiken zeichnen will. Zeichnet man einen Kreis im Raum, auf dessen Fläche man nicht genau senkrecht blickt, so wird er zum Oval. Und Kreisbögen werden genau in der Art und Weise verzerrt, wie es QuickDraw tut.

3.5.6 Polygone

Mit Polygonen bezeichnet die QuickDraw-Dokumentation eine endliche Folge von Punkten, die untereinander durch gerade Linien verbunden sind. Die

Folge der Punkte kann dabei (nahezu) beliebig lang sein. Man könnte ein Polygon auch als eine gespeicherte Folge von **LineTo**-Befehlen auffassen. Da die Größe der Speicherstruktur, die ein Polygon bestimmt, abhängig von der Zahl der Punkte ist und diese zunächst unbekannt ist, werden diese Strukturen wieder einmal als variabler verschieblicher Speicherblock, als **Handle**, abgelegt. Die entsprechenden Typvereinbarungen sehen aus wie folgt:

```
TYPE PolyHandle = ^PolyPtr;
     PolyPtr    = ^Polygon;
     Polygon    =
        RECORD
            polySize   : INTEGER;
            polyBBox   : Rect;
            polyPoints : ARRAY [0..32000] OF Point;
        END{RECORD Polygon};
```

Im Array **polyPoints** werden die einzelnen Punkte, zwischen denen das Polygon gebildet wird, gespeichert. Natürlich ist dieses Array nicht wirklich 32001 Elemente groß, sondern immer nur so groß, wie es für ein bestimmtes Polygon nötig ist. Da man sich dynamisch ändernde Arrays aber in Pascal nicht ausdrücken kann, wurde die Konstruktion mit dem nahezu beliebig langen Array gewählt. Die beiden andern Felder des Records enthalten die wirkliche Länge der Speicherstruktur (im Feld **polySize**) und das kleinste Rechteck, in dem alle Punkte liegen (im Feld **polyBBox**). Diese Struktur beschreibt vollständig das Aussehen des Polygons, deshalb sehen die Operationen mit Polygonen auch wie folgt aus:

```
PROCEDURE FillPoly   (poly: PolyHandle;
                      pat: Pattern);
PROCEDURE PaintPoly  (poly: PolyHandle);
PROCEDURE ErasePoly  (poly: PolyHandle);
PROCEDURE InvertPoly (poly: PolyHandle);
PROCEDURE FramePoly  (poly: PolyHandle);
```



Bild 3 - 20: *Die Wirkung von FramePoly*

Eine Besonderheit hat allerdings die Operation **FramePoly**. Wie ich oben schon erwähnt habe, entspricht die Datenstruktur eines Polygons in etwa einer gespeicherten Folge von **LineTo**-Aufrufen. Genau diese Folge von Aufrufen läuft dann bei **FramePoly** wieder ab. Dies hat zur Folge, daß die Bits, die von dieser Operation beeinflußt werden, auch außerhalb der vom Polygon umschlossenen Fläche liegen können; ja sogar außerhalb von **polyBBox**. Die ist anders als bei allen anderen **Frame**...-Operationen und deshalb leicht eine Ursache von unerwarteten Effekten am Bildschirm. Der Grund für dieses merkwürdige Verhalten liegt in der Realisierung der ...**Poly**-Operationen, die so viel schneller und effizienter ausfallen konnten.

Es wäre nun sehr umständlich, die **PolyHandle**-Struktur "von Hand" erzeugen zu müssen und selbst alle Punkt-Koordinaten in das Array eintragen zu müssen, dessen Länge ja dadurch jedesmal wächst. Deshalb sieht QuickDraw eine Möglichkeit vor, eine Folge von **LineTo**-Aufrufen direkt in einem Polygon zu speichern. Hierzu dienen die folgenden Prozeduren:

```
FUNCTION   OpenPoly  : PolyHandle;
PROCEDURE  ClosePoly;
PROCEDURE  KillPoly (poly: PolyHandle);
```

OpenPoly ist die Mitteilung an QuickDraw, daß man nun das Zeichnen eines Polygons beginnen wird. Gleichzeitig wird als Funktionswert der entsprechende **PolyHandle** des, jetzt noch "leeren", Polygons zurückgeliefert. QuickDraw selbst speichert das aktuelle Polygon im Feld **savePoly** des aktuellen GrafPorts. Nach dem Aufruf von **OpenPoly** kann der Programmierer mit einer beliebig langen Folge von **LineTo**-Befehlen das Polygon zeichnen. Wenn er damit fertig ist, muß er unbedingt **ClosePoly** aufrufen, um QuickDraw mitzuteilen, daß das Polgon beendet ist. Daraufhin berechnet QuickDraw noch das Feld **polyBBox** des Polygons und entfernt den **PolyHandle** aus **savePoly**.

Erst nach **ClosePoly** haben Zeichenbefehle wieder eine Wirkung auf dem Bildschirm. Mit **OpenPoly** wird nämlich nicht nur ein neues leeres Polygon erzeugt, sondern auch das Feld **pnVis** des GrafPorts um 1 kleiner gemacht. Da es normalerweise den Wert 0 hat, ist es danach gleich –1. Dies führt dazu, das alle QuickDraw-Befehle evtl. aufgezeichnet werden, wie z.B. die **LineTo**-Befehle, aber keine Wirkung auf den Bildschirm (oder eine andere BitMap) haben. Eine mögliche Befehlsfolge zur Erstellung eines Dreiecks, das QuickDraw ansonsten nicht als eigenständigen Objekt-Typ kennt, sähe z.B. wie folgt aus:

```
dreiEck := OpenPoly;
         MoveTo (100,0);
         LineTo (50,100);
         LineTo (150,100);
         LineTo (100,0);
ClosePoly;
FillPoly(dreiEck,black);
```

Ein Polygon benötigt unter Umständen beachtlichen Speicherplatz — 4 Byte für jeden Punkt plus insgesamt 22 Bytes Verwaltungsinfomation. Deshalb ist es ratsam, die Polygon-Speicherstruktur zu löschen, sobald das Polygon nicht mehr benötigt wird. Dies geschieht mit **KillPoly**.

3.5.7 Regionen

Regionen sind Polygonen recht ähnlich. Während ein Polygon allerdings durch eine Folge von Punkten bzw. Linien beschrieben wird, die man QuickDraw mit einer Folge von **LineTo**-Befehlen mitteilt, können Regionen wesentlich komplexer aussehen. Sie können prinzipiell jede beliebige, wie auch immer geformte, Fläche in der Grafikebene darstellen und deshalb auch jede beliebige Menge von Pixeln in der BitMap beschreiben. Ihre genaue Struktur kann auch von Pascal aus überhaupt nicht mehr beschrieben werden und ist auch nicht dokumentiert. Regionen sind sehr wichtig für die Fensterverwaltung und bieten zusätzlich dazu noch die Möglichkeit, komplexe Formen mit nur einem Prozeduraufruf zeichnen zu können.

Die Operationen, die mit Regionen möglich sind, sind ansonsten dieselben, wie sie z.B. für Rechtecke oder Polygone zur Verfügung stehen. Sie beeinflussen allerdings, anders als **FramePoly**, immer nur die in der Region liegenden Pixel.

```
PROCEDURE FillRgn    (rgn: RgnHandle;
                      pat: Pattern);
PROCEDURE PaintRgn   (rgn: RgnHandle);
PROCEDURE EraseRgny  (rgn: RgnHandle);
PROCEDURE InvertRgn  (rgn: RgnHandle);
PROCEDURE FrameRgn   (rgn: RgnHandle);
```



Bild 3 - 21:*Die Wirkung von FrameRgn*

Die Erzeugung und Veränderung von Regionen ist natürlich noch ein größeres Problem als die Behandlung von Polygonen. Hierzu stehen deshalb die folgenden Operationen zur Verfügung:

```
FUNCTION  NewRgn  : RgnHandle;
PROCEDURE OpenRgn;
PROCEDURE CloseRgn    (rgn: RgnHandle);
PROCEDURE DisposeRgn  (rgn: RgnHandle);
PROCEDURE SetRectRgn
            (rgn: RgnHandle;
             left, top, right, bottom: INTEGER);
```

NewRgn erzeugt eine neue leere Region und liefert deren Handle zurück. **OpenRgn** teilt QuickDraw mit, daß die jetzt folgenden Zeichen-Befehle zur Definition einer Region verwendet werden sollen und nicht auf dem Bildschirm erscheinen sollen. Hierzu wird wieder **pnVis** um 1 herabgesetzt. Alle folgenden Zeichen-Befehle, die eine geschlossene Fläche zeichnen, erweitern dann die aktuelle Region um diese Fläche. **CloseRgn** schließlich beendet die Definition einer Region, berechnet **rgnBox** und kopiert die bisher in **rgnSave** gehaltene Datenstruktur in den Parameter **rgn.pnVis**, wird danach wieder um 1 heraufgesetzt. **SetRectRgn** schließlich ist eine

einfachere Möglichkeit, eine rechteckige Region zu erzeugen. Rechteckige Regionen werden häufig für die Fensterverwaltung benötigt.

Das folgende Programmstück erzeugt und zeichnet z.B. eine hantelförmige Region:

```
hantel := NewRgn;
OpenRgn;
        SetRect(temp, 0,0,100,100);
        FrameOval(temp);          { erster Kreis }
        SetRect(temp, 30,30,320,70);
        FrameRect(temp);          { Verbindungsbalken }
        SetRect(temp, 300,0,400,100);
        FrameRect(temp);          { zweiter Kreis }
CloseRgn(hantel);
PaintRgn(hantel);
```

Die folgende Zeichnung verdeutlicht diesen Vorgang. Die gepunkteten Linien darin zeigen die Flächen, aus denen sich die Region zusammensetzt. Die gestrichelte Linie entspricht dem Wert des Feldes **rgnBox** im Region-Record.

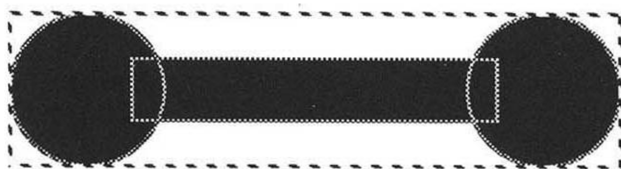


Bild 3 - 22: *Die Definition einer Region*

Da Regionen sogar noch mehr Speicherplatz als Polygone benötigen, sollten auch sie sobald wie möglich mit **DisposeRgn** gelöscht werden.

QuickDraw bietet noch eine Reihe weiterer Möglichkeiten, das Aussehen von Regionen zu verändern. Es handelt sich dabei im wesentlichen um Mengen-Operationen. Regionen können vereinigt und geschnitten werden und vieles mehr. Ich werde auf diese, meist nicht benötigten, Operationen aber hier nicht näher eingehen und erwähne sie nur der Vollständigkeit halber.

3.5.8 Zusammengesetzte Bilder

Wie wir ja schon gesehen haben, kann man Polygone als gespeicherte Folge von **LineTo**-Befehlen auffassen — insbesondere in bezug auf den Befehl **FramePoly**, der diese gespeicherte Befehlsfolge wieder abarbeiten kann. QuickDraw kennt aber noch eine viel mächtigere Möglichkeit, Befehlsfolgen zu speichern. Bilder (engl. *pictures*) sind eine kompakte Repräsentation einer (nahezu) beliebigen Folge von QuickDraw-Funktionsaufrufen. Sie werden in einer Datenstruktur der folgenden Form gespeichert:

```
TYPE PicHandle   = ^PicPtr;
     PicPtr      = ^Picture;
     Picture     =
        RECORD
            picSize   : INTEGER;
            picFrame  : Rect;
            picData   : ARRAY [0..32000] OF Byte;
        END{RECORD Polygon};
```

picSize enthält dabei die wirkliche Größe der Datenstruktur in Bytes, und **picFrame** enthält ein Rechteck, das das Bild vollständig umschließt. Ähnlich wie bei Polygonen, sind die wichtigsten Aspekte dieser Struktur (in **picData** verborgen) nicht von Pascal aus erreichbar. Diese Struktur ist auch nicht dokumentiert. Alle mit Bildern möglichen Manipulationen beschränken sich im wesentlichen darauf, Bilder zu erzeugen, zu zerstören und zu zeichnen.

```
FUNCTION  OpenPicture  (picFrame : Rect)
           : PicHandle;
PROCEDURE DrawPicture
           (pic: PicHandle;
            dstRect: Rect);
PROCEDURE ClosePicture;
PROCEDURE KillPicture  (pic: PicHandle);
```

Die Funktion **OpenPicture** liefert den Handle eines zunächst noch leeren Bildes zurück und teilt QuickDraw mit, daß die nun folgenden Zeichenoperationen gespeichert werden sollen. Ähnlich den Befehlen **OpenRgn** und **NewPoly** wird **pnVis** im GrafPort um 1 herabgesetzt und das augenblicklich definierte Bild im Feld **picSave** des Grafports gespeichert. Anders als bei **OpenRgn** oder **NewPoly** muß man **OpenPicture** allerdings ein Rechteck mitteilen, das das im folgenden gezeichnete Bild vollständig umschließt. Dieses Rechteck wird auch im Feld **picFrame** des Pictures gespeichert.

Nach **OpenPicture** kann man eine Folge von Zeichenbefehlen aufrufen, mit denen man das zu speichernde Bild auf dem Bildschirm zeichnen würde. Ist diese Folge beendet — von der ja bis jetzt am Bildschirm nichts zu sehen ist — ruft man **ClosePicture** auf. Daraufhin hört QuickDraw auf, Operationen zu speichern und setzt **pnVis** wieder um 1 herauf; Zeichnungen gehen jetzt wieder auf den Bildschirm. In dem Handle, das **OpenPicture** zurückgegeben hatte, stehen jetzt alle Infomationen, die QuickDraw benötigt, um diese Folge von Zeichenbefehlen zu wiederholen. Mit dem Befehl **DrawPicture** kann man nun diese ganze Folge von Befehlen durch einen einzigen Funktionsaufruf ablaufen lassen. **DrawPicture** benötigt wie **OpenPicture** die Angabe eines Rechtecks **dstRect**. QuickDraw verschiebt, staucht und dehnt beim Zeichnen eines Pictures automatisch alle Koordinaten, damit das Bild, das in **picFrame** gezeichnet wurde, nun in das neue Rechteck paßt.

Die folgende Zeichnung versucht, diesen Vorgang zu verdeutlichen. Ich erspare mir aus Platzgründen allerdings, das entsprechende Pascal-Programmstück hier aufzulisten, das dieses Bild erzeugt hat.

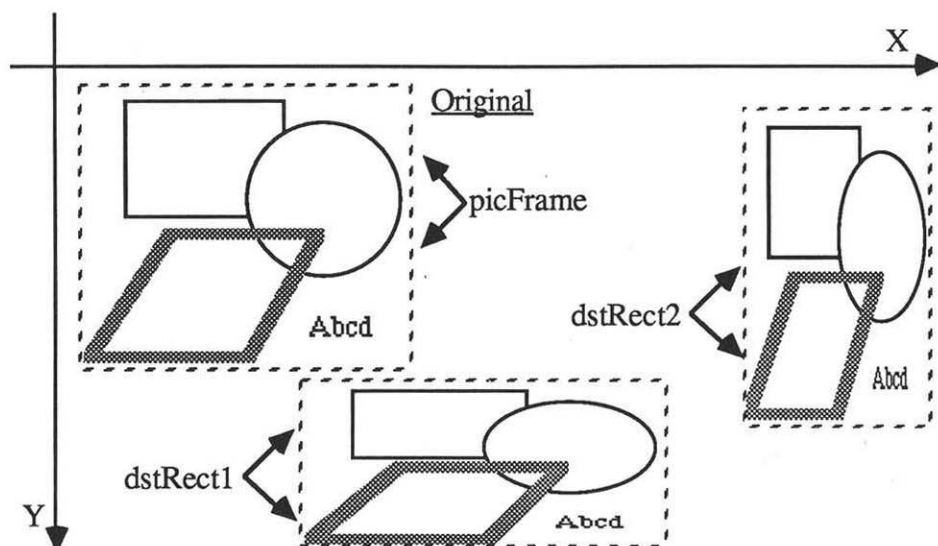


Bild 3 - 23: *Das Zeichnen eines gespeicherten Bildes*

Wie man sehr gut sehen kann, sind Pictures wirklich gespeicherte Zeichenbefehle. Die Muster und Texte werden z.B. nicht verzerrt, wenn das Bild in ein wesentlich kleineres Rechteck gezeichnet wird. Nur die Koordinaten der Rechtecke und Punkte, die man beim Zeichnen des Bildes angegeben hat, werden verändert.

Pictures benötigen leider einen ganz beachtlichen Platz. Alle Zeichenbefehle, Punkte, Rechtecke etc. müssen ja irgendwie gespeichert werden. Dies kann für kompliziertere Zeichnungen recht groß werden. Es ist z.B. auch möglich, komplette BitImages in Pictures abzulegen, die oft schon für recht kleine Bilder viel Platz benötigen. Obwohl diese Möglichkeit der Bildspeicherung sehr praktisch ist, sollte man deshalb sparsam damit umgehen und sobald als möglich das Bild wieder mit **KillPicture** löschen.

Pictures sind übrigens auch die Datenstruktur, über die z.B. MacPaint und MacDraw mit MacWrite Abbildungen austauschen können. Da das Format universell ist und seine Kodierung/Dekodierung im ROM des Macintosh passiert (und das Programm nicht weiter zu kümmern braucht), ist es sehr leicht, Bilder zwischen Programmen auszutauschen. Alles, was das "empfangende" Programm tun muß, ist, genügend Platz für das Bild zu

finden und evtl. zu reservieren, damit das Bild nicht mit den eigenen Ausgaben — z.B. Texten — in Konflikt tritt.

MacWrite (oder auch MS-WORD) bietet auch dem Benutzer z.B. noch die Möglichkeit an, dstRect nachträglich zu verändern. Klickt man in diesem Programm eine Abbildung an, so erscheinen an deren Rand kleine schwarze Quadrate, die man mit der Maus ergreifen und so die Abmessungen des Bildes und seine Proportionen total verändern kann.

3.6 Kalkulationen mit grafischen Objekten

Die folgende Gruppe von Funktionen zeichnet überhaupt nichts auf den Bildschirm oder in eine andere BitMap. Es handelt sich dabei um Kalkulations-Operationen, die grafische Objekte wie Punkte, Rechtecke oder Regionen untersuchen und manipulieren können. Sie erlauben z.B. die Feststellung, ob sich ein gewisser Punkt noch innerhalb eines Rechtecks oder einer bestimmten Region befindet. Sie berechnen Schnittflächen zweier Rechtecke, die zwei gegebene Punkte gerade noch enthalten etc. Die Bedeutung dieser Gruppe von Funktionen liegt vor allem in der Untersuchung von Mausklicks und in bewegter Grafik. Für Computerspiele ist es ja z.B. sehr wichtig, festzustellen, welche Objekte sich auf dem Bildschirm berührt haben oder welche Gebiete ein bestimmtes Objekt überdeckt. Und Mausklicks können sehr unterschiedliche Wirkungen haben, je nachdem, in welche Gebiete sie fallen.

Da QuickDraw aber ein sehr reichhaltiges Angebot von Funktionen und Prozeduren hat, die grafische Objekte prüfen und manipulieren, stelle ich hier nur eine kleine Auswahl davon vor. Hat der Leser aber erst einmal eine Vorstellung von der Funktionsweise dieser Operationen bekommen, wird er die restlichen — die seltener benötigt werden — ohne Schwierigkeiten mit Hilfe der Dokumentation seiner Programmiersprache verstehen können.

3.6.1 Kalkulationen mit Punkten

Die Routinen von QuickDraw dienen dazu, den Umgang mit Punkten etwas zu vereinfachen. Sie machen Programme kürzer und lesbarer. Sie sind fast alle recht trivial. D.h., daß sie alle von jedem einigermaßen erfahrenen Programmierer in kürzester Zeit und in wenigen Programmzeilen selbst realisiert werden könnten. Die beiden folgenden Beschreibungen stellen zwei der häufiger gebrauchten Routinen aus dieser Gruppe vor.

```
PROCEDURE SetPt(VAR pt: Point; h,v: INTEGER);
```

SetPt ist mehr oder weniger nur eine Abkürzung für das Setzen der beiden Punkt-Koordinaten von Hand: `pt.h := h; pt.v := v;`

```
FUNCTION EqualPt(pt1,pt2: Point): BOOLEAN;
```

Die Funktion **EqualPt** ergibt genau dann TRUE, wenn sowohl die horizontale wie auch die vertikalen Koordinaten der beiden Punkte übereinstimmen.

3.6.2 Kalkulationen mit Rechtecken

Genau wie die Kalkulationen mit Punkten dienen auch die Prozeduren und Funktionen zu Kalkulationen mit Rechtecken hauptsächlich der Bequemlichkeit des Programmierers und der Übersichtlichkeit des Programms. Die meisten dieser Prozeduren könnten in wenigen Zeilen auch selbst geschrieben werden.

```
PROCEDURE SetRect(VAR r: Rect;
                  left,top,right,bottom : INTEGER);
```

SetRect ist eine einfache Methode, ein Rechteck zu positionieren, die wesentlich kürzer und lesbarer ist, wie die einzelnen Felder der Records nacheinander zu belegen. Sie entspricht dem Programmstück

```
r.left      := left;
r.top       := top;
r.right     := right;
r.bottom    := bottom;
```

```
FUNCTION EmptyRect(r: Rect): BOOLEAN;
```

EmptyRect ergibt TRUE, wenn **r** keine Fläche auf der Grafikebene umschließt. Dies ist dann der Fall, wenn der obere linke Eckpunkt von **r** gleich dem unteren rechten ist oder wenn **r.botRight** sogar oberhalb oder links von **r.topLeft** liegt.

```
FUNCTION EqualRect(r1,r2: Rect): BOOLEAN;
```

Diese Funktion ist eine Kurzschreibweise für:

```
EqualPt(r1.topLeft,r2.topLeft)AND
EqualPt(r1.topLeft,r2.topLeft)
```

```
FUNCTION PtInRect(pt: Point; r: Rect): BOOLEAN;
```

PtInRect prüft, ob der Punkt **pt** im Innern oder auf den Kantenlinien des Rechtecks **r** liegt, und ergibt in diesem Fall TRUE. **PtInRect** liefert FALSE, falls **pt** außerhalb von **r** liegt.

Die folgenden Prozeduren dienen zum Verschieben und Verändern von Rechtecken und finden natürlich auch besonders bei bewegten Grafiken und ähnlichen Anwendungen Verwendung.

```
PROCEDURE OffsetRect(VAR r: Rect; dH,dV: INTEGER);
```

Die Prozedur OffsetRect dient zum Verschieben des Rechtecks **r** um **dH** Punkte nach rechts (d.h. nach links, falls **dH** negativ ist) und **dV** Punkte nach unten (d.h. nach oben, falls **dV** negativ ist).

```
PROCEDURE InsetRect(VAR r: Rect; dH,dV: INTEGER);
```

InsetRect läßt das Rechteck **r** schrumpfen oder wachsen, wobei das Zentrum am gleichen Punkt verbleibt. Positive Beträge der Parameter **dH** und **dV** lassen das Rechteck schrumpfen und negative dehnen es aus. Der Aufruf der Prozedur **InsetRect** entspricht dem folgenden Programmstück:

```
WITH r DO
    BEGIN
    left        := left+dH;
    top         := top+dV;
    right       := right-dH;
    bottom      := bottom-dV;
    END;
```

Umschließt **r** nach dieser Kalkulation keine Fläche auf der Grafikebene mehr — ist **EmptyRect(r)** also gleich TRUE — so wird es noch vor Verlassen von **InsetRect** auf das leere Rechteck ((0.0),(0,0)) gesetzt.

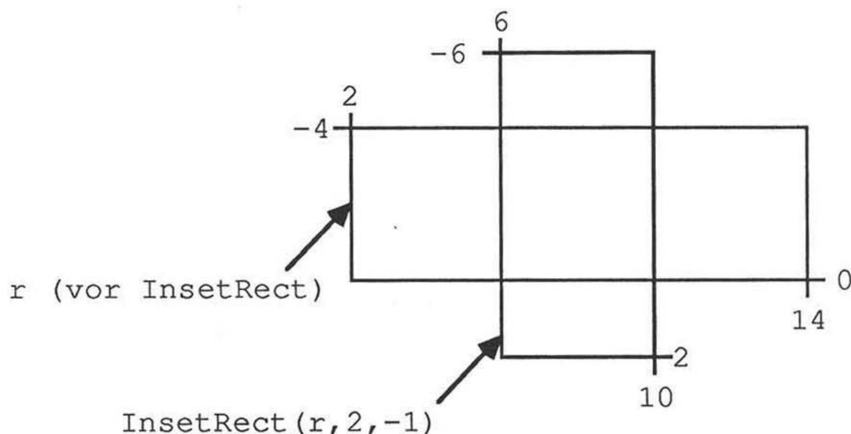


Bild 3 - 24: *Die Wirkung von InsetRect*

```
PROCEDURE SectRect(src1,src2: Rect;
                   VAR dstRect: Rect);
```

SectRect (nicht zu verwechseln mit **SetRect**) berechnet in **dstRect** die Schnittfläche von **src1** und **src2**. dstRect enthält also nach Aufruf dieser Prozedur dasjenige Rechteck, das die Fläche umschließt, die sowohl innerhalb von **src1** wie auch von **src2** liegt. Überschneiden **src1**und **src2t** sich nicht, wird **dstRect** auf ((0,0),(0,0)) gesetzt.

```
PROCEDURE UnionRect(src1,src2: Rect;
                    VAR dstRect: Rect);
```

UnionRect berechnet in **dst** das kleinste Rechteck, das **src1** und **src2** gerade noch enthält.

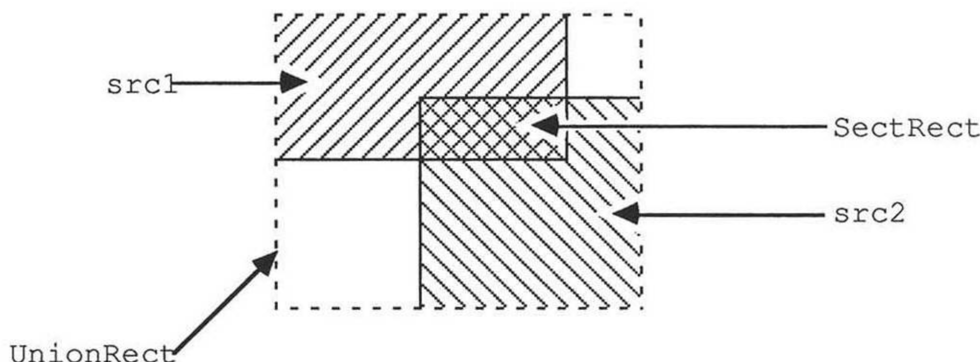


Bild 3 - 25: *UnionRect und SectRect am Beispiel*

```
PROCEDURE Pt2Rect(pt1,pt2: Point;
                  VAR dstRect: Rect);
```

Pt2Rect berechnet das kleinste Rechteck, das **pt1** und **pt2** gerade noch enthält. Dies ist natürlich das Rechteck mit **pt1** und **pt2** als Eckpunkten. Es ergibt nur im Programm eine ziemlich komplizierte Schachtel von IF...THEN...ELSE Anweisungen, wenn man feststellen will, welcher der beiden Punkte die linke obere und welches die rechte untere Ecke darstellt. **Pt2Rect** nimmt einem diese Arbeit ab. Sind **pt1** und **pt2** gleich, so wird dstRect auf ((0,0),(0,0)) gesetzt.

3.6.3 Kalkulationen mit Regionen

Während die meisten Kalkulationen, die QuickDraw einem durch fertige Operationen für Rechtecke und Punkte abnimmt, eher trivial sind, werden die folgenden Operationen, die dasselbe für Regionen leisten, dringend benötigt. Regionen sind recht komplexe Datenstrukturen, deren genaue Struktur zudem nicht dokumentiert ist. Wenn irgendwelche Veränderungen daran vorgenommen werden sollen oder etwas über ihr Aussehen festgestellt werden soll, bleibt einem deshalb nichts anderes übrig, als die folgenden Funktionen und Prozeduren zu verwenden.

```
FUNCTION EmptyRgn(rgn: RgnHandle): BOOLEAN;
```

EmptyRgn ergibt TRUE, falls **rgn** die leere Fläche enthält, d.h. keine Pixel der BitMap in ihr enthalten sind.

```
FUNCTION EqualRgn(rgn1,rgn2: RgnHandle): BOOLEAN;
```

EqualRgn ergibt TRUE, falls beide Regionen exakt dieselben Gebiete umschließen, d.h. die beiden Datenstrukturen am Ende der beiden Handles sich exakt gleichen. Leere Regionen sind immer gleich!

```
FUNCTION RectInRgn(r: Rect;rgn: RgnHandle)
           : BOOLEAN;
```

RectInRgn stellt fest, ob sich die gesamte Fläche des Rechtecks **r** innerhalb der Region **rgn** befindet, und liefert nur in diesem Fall TRUE.

```
FUNCTION PtInRgn(r: Rect;rgn: RgnHandle): BOOLEAN;
```

PtInRgn stellt fest, ob sich der Punkt **p** innerhalb der Region **rgn** befindet und liefert nur in diesem Fall TRUE.

```
PROCEDURE SetEmptyRgn(rgn: RgnHandle);
```

Die Prozedur **SetEmptyRgn** setzt den RgnHandle **rgn** auf die leere Region. **EmptyRgn(rgn)** ergibt danach natürlich TRUE und das Feld **rgnBBox** der RegionRecords enthält ((0,0),(0,0)).

```
PROCEDURE RectRgn(rgn: RgnHandle; r: Rect);
```

Nach dem Aufruf von **RectRgn** enthält **rgn** genau die Fläche, die vom Rechteck **r** umschlossen wird. Das Feld **rgnBBox** des RegionRecords ist nach dem Aufruf von **RectRgn** gleich **r**.

```
PROCEDURE OffsetRgn(rgn: RgnHandle;
           dH,dV: INTEGER);
```

OffsetRgn verschiebt die Region **rgn** um **dH** Punkte nach rechts (bzw. nach links, wenn **dH** negativ ist) und **dV** Punkte nach unten (bzw. nach oben, wenn **dV** negativ ist). Dies entspricht einem Aufruf von OffsetRect für ein Rechteck. Die Form der Region bleibt bei dieser Verschiebung absolut gleich. Leere Regionen werden durch Verschieben in keinster Weise modifiziert.

```
PROCEDURE InsetRgn(rgn: RgnHandle;
                   dH,dV: INTEGER);
```

InsetRgn entspricht der Prozedur **InsetRect** für Rechtecke. Die Wirkung auf die Region **rgn** läßt sich allerdings sehr schwer in Worte fassen. Prinzipiell kann nur gesagt werden, daß die Region für positive Werte von **dH** bzw. **dV** horizontal bzw. vertikal schrumpft und sich für negative Werte ausdehnt. Dabei kann **rgn** zur leeren Region werden.

```
PROCEDURE SectRgn(srcRgnA,srcRgnB,
                  dstRgn: RgnHandle);
```

Nach dem Aufruf von **SectRgn** enthält **dstRgn** genau die Pixel, die sowohl in der Region **srcRgnA** wie auch in **srcRgnB** liegen. Gibt es keine Pixel, die sowohl in **srcRgnA** wie auch in **srcRgnB** liegen, wird **dstRgn** auf die leere Region gesetzt. Dies entspricht mathematisch dem Schnitt der beiden Regionen (Mengen von Pixeln).

```
PROCEDURE UnionRgn(srcRgnA,srcRgnB,
                   dstRgn: RgnHandle);
```

Nach dem Aufruf von **UnionRgn** enthält **dstRgn** genau die Pixel, die in **srcRgnA** oder in **srcRgnB** liegen. Dies entspricht mathematisch der Vereinigung der beiden Regionen (Mengen von Pixeln).

3.6.4 Globale und lokale Koordinatensysteme

QuickDraw bietet die Möglichkeit — ja fordert geradezu dazu auf — viele verschiedene GrafPorts für unterschiedliche Zwecke zu verwenden, zwischen denen man schnell und problemlos hin und her wechseln kann. Nicht ganz problemlos allerdings! Jeder GrafPort definiert ja durch seine BitMap ein eigenes Koordinatensystem auf seinem BitImage.

Nehmen wir an, dieses BitImage wäre der Bildschirmpuffer des Macintosh (eine wohl häufig zutreffende Annahme). Ein und dieselbe Position auf dem Bildschirm kann in einem GrafPort die Koordinaten (0,0) haben und in einem anderen die Koordinaten (–24395,7318). Meist ist es zwar für ein Programm völlig uninteressant, zu wissen, welche Beziehung zwischen zwei Positionen in unterschiedlichen GrafPorts besteht, manchmal wird es aber wichtig.

Zeigen z.B. ein Dutzend verschiedene GrafPorts auf den Bildschirmpuffer, und ein Programm fragt die Mausposition ab, so ist fraglich, welcher

GrafPort für die Angabe der Koordinaten der Maus-Position verwendet wird. Es ist nicht unbedingt immer sinnvoll, dafür den aktuellen GrafPort zu nehmen.

Wer glaubt, daß es eine sehr seltene Situation wäre, zwischen einem Dutzend GrafPorts hin und her zu wechseln, der möge sich vor Augen halten, daß jedes Fenster ein eigener GrafPort ist. Und mit Fenstern geht der Macintosh nicht gerade sparsam um. Die meisten dieser Fenster definieren üblicherweise ihre obere linke Ecke als den Punkt (0,0) in ihrem GrafPort und legen so völlig unterschiedliche Koordinatensysteme über ein und dieselbe BitMap — den Bildschirmpuffer.

Um diesen Konflikt zu bewältigen, gibt es für jedes BitImage ein globales Koordinaten-System. In diesem Koordinatensystem liegt die obere linke Ecke des Rechtecks **bounds** der entsprechenden BitMap immer bei (0,0). Koordinaten in jedem GrafPort, der auf diesem BitImage arbeitet, können in das globale Koordinatensystem umgewandelt werden und wieder zurück ins lokale Koordinatensystem des GrafPorts.

Sollten diese Ausführungen Sie, den Leser, jetzt verwirrt haben, kann es nicht schaden, sich noch einmal genau die Beschreibung der Zusammenhänge zwischen BitImages, BitMaps und GrafPorts zu Beginn dieses Kapitels durchzulesen. Insbesondere sollte man sich über die Funktion des Feldes **bounds** einer BitMap klarwerden.

```
PROCEDURE LocalToGlobal(VAR pt: Point);
```

LocalToGlobal berechnet die Koordinaten, die der Punkt **pt** im globalen Koordinatensystem der BitMap einnimmt. **LocalToGlobal** nimmt an, daß die augenblicklichen Koordinaten von **pt** im Koordinatensystem des aktuellen GrafPorts gemessen wurden. Würde man **LocalToGlobal** selbst schreiben, würde es in etwa wie folgt aussehen:

```
PROCEDURE LocalToGlobal(VAR pt: Point);
BEGIN
   pt.h := pt.h -
               thePort^.portBits.bounds.topLeft.h;
   pt.v := pt.v -
               thePort^.portBits.bounds.topLeft.v;
END;
```

ThePort ist dabei stets der aktuelle GrafPort. Das BitImage dieses GrafPorts wird zumeist der Bildschirmpuffer selbst sein.

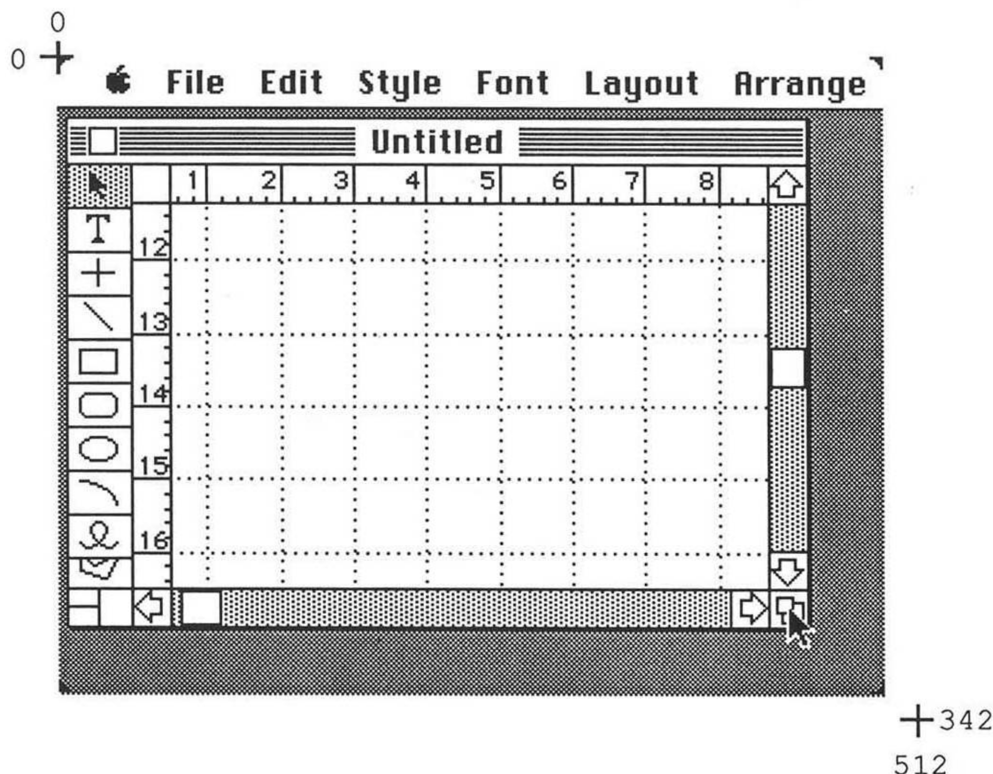


Bild 3 - 26: *Das globale Koordinatensystem des Bildschirmpuffers*

PROCEDURE `GlobalToLocal(VAR pt:  Point);`

GlobalToLocal kehrt die Wirkung von **LocalToGlobal** genau um. Die Prozedur nimmt an, daß die Koordinaten von **pt** im globalen Koordinatensystem des aktuellen GrafPorts betrachtet werden müssen und berechnet daraus die lokalen Koordinaten desselben Punktes in der Grafikebene, wenn er im lokalen Koordinatensystem des aktuellen GrafPorts betrachtet wird.

Wollte man z.B. feststellen, ob zwei Rechtecke, die in zwei unterschiedlichen GrafPorts **port1** und **port2** gezeichnet wurden, auf dem Bildschirm eine gemeinsame Schnittfläche haben, so würde man dafür das folgende Programmstück verwenden.

```
SetPort(port1)
LocalToGlobal(rect1.topLeft);
LocalToGlobal(rect1.botRight);
SetPort(port2)
LocalToGlobal(rect2.topLeft);
LocalToGlobal(rect2.botRight);
IF SectRect(rect1,rect2,rect3) THEN
     ...............
```

Der Leser möge sich die Wirkungsweise von **GlobalToLocal** und **LocalToGlobal** noch einmal genau verdeutlichen, bevor er weiterliest. Das Verständnis dieser Konzepte ist elementar für einige der folgenden Kapitel. Man muß sich z.B. auch klarmachen, daß GrafPorts, die auf verschiedenen BitImages arbeiten (z.B. die eine auf dem Bildschirm und die andere auf einem unsichtbaren Puffer), kein gemeinsames globales Koordinatensystem besitzen können. Das globale Koordinatensystem eines GrafPorts dient ja hauptsächlich dazu, zu Pixeln, die in bestimmten Flächen eines Koordinatensystems liegen, die entsprechende Fläche in einem anderen Koordinatensystem zu finden. Zwischen Pixeln (Bits), die in völlig verschiedenen BitImages liegen, kann es logischerweise aber keine sinnvolle Beziehung in dieser Form geben.

3.7 Ein Beispielprogramm zu QuickDraw

Zur Verdeutlichung einiger der komplizierteren Konzepte von QuickDraw sollen diese nun anhand eines kleinen Beispielprogramms vorgeführt werden. Dieses Programm dient nur als Beispiel zu QuickDraw und nutzt deshalb außer der Grafik kaum irgendwelche der besonderen Eigenschaften des Macintosh (wie etwa PullDown-Menüs und Fenster).

3.7.1 Überblick über das Beispielprogramm

Das Programm *ShowQuickDraw* (Zeige QuickDraw) dient hauptsächlich zur Demonstration von QuickDraw-Pictures (gespeicherten Bildern, die sich aus vielen einzelnen Zeichen-Operationen zusammensetzen) und Regionen insbesondere der **clipRgn** des aktuellen GrafPorts, die sämtliche Zeichenausgaben auf ein bestimmtes Gebiet des Bildschirms begrenzt.

Hierzu speichern wir am Anfang ein kleines Bild in einer globalen Variablen des Programms. Dieses Bild wird dann während des Programmlaufs immer dem Maus-Cursor folgen. Und zwar wird die linke untere Ecke des Bildes stets an der Stelle sein, wo sich auch der Maus-Cursor auf dem Bildschirm befindet. Das Zeichnen dieses gespeicherten Bildes wird aber immer durch die aktuelle clipRgn des GrafPorts begrenzt, die wir am Anfang des Programms auf einen Bereich setzen, der aus zwei Schlüsselloch-förmigen Gebieten auf dem Bildschirm besteht. Die folgende Abbildung zeigt ein typisches Aussehen des Bildschirm während das Programm läuft. Wie gut zu sehen ist, hört das Zeichnen des Bildes (innerhalb des dünnen Rechtecks) schlagartig an der Grenze auf, die durch die beiden hellgrau umrandeten Schlüssellöcher gebildet wird.

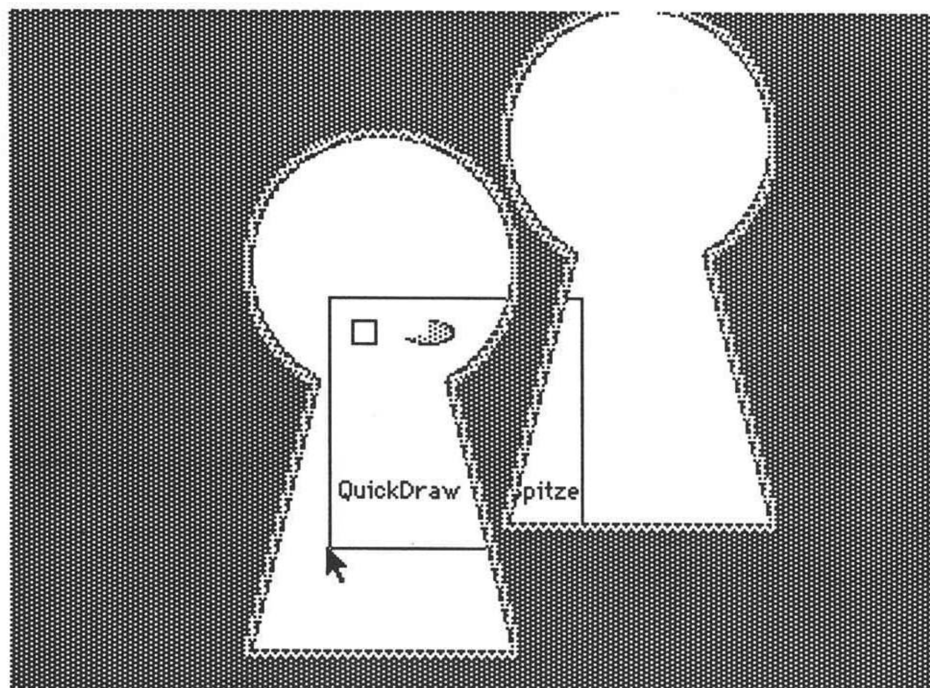


Bild 3 - 27: *Ein Beispiel zur Verwendung von QuickDraw*

3.7.2 Das Hauptprogramm

Zum besseren Verständnis des Programms wird es "auf den Kopf" gestellt. D.h.: anders als ein Pascal-Compiler, der jede Prozedur zunächst definiert sehen möchte, bevor sie im Programm verwendet werden kann, wird zunächst das Hauptprogramm vorgestellt und danach die einzelnen aufgerufenen Prozeduren erläutert. Dies nennt man, wenn man es bei der Programmierung macht und nicht nur bei der Erläuterung des Programms, oft auch "Top-Down-Vorgehensweise" oder das "Prinzip der schrittweisen Verfeinerung".

```
PROGRAM ShowQuickDraw;
USES ......, QuickDraw,etc. ; { in UCSD-Pascal }
VAR
    alterPunkt,
    neuerPunkt:         Point;

    QDPict:             PicHandle;
    clip:               RgnHandle;

    port:               GrafPort;


 PROCEDURE InitProg;
 BEGIN
    ............ { wird weiter unten beschrieben }
 END; { InitProg }

 PROCEDURE ZeichneBild(bild: PicHandle; pt: Point);
 BEGIN
    ............ { wird weiter unten beschrieben }
 END; { ZeichneBild }
```

```
BEGIN { ShowQuickDraw }
    InitProg;
    ZeichneBild(QDPict,alterPt);
    REPEAT
        GetMouse(neuerPunkt);
        IF NOT EqualPt(neuerPunkt,alterPunkt) THEN
            BEGIN
            ZeichneBild(QDPict,alterPunkt);
            ZeichneBild(QDPict,neuerPunkt);
            alterPunkt := neuerPunkt;
            END;
    UNTIL Button;
END.  { ShowQuickDraw }
```

Betrachten wir uns das Hauptprogramm einmal näher: Zu Beginn werden erst einmal durch den Aufruf von **InitProg** alle globalen Variablen gesetzt und der Bildschirm (der GrafPort) für das folgende Programm vorbereitet. **InitProg** wird im nächsten Abschnitt noch näher beschrieben. Danach wird das in QDPict gespeicherte QuickDraw-Picture ein erstes Mal gezeichnet.

Die nun folgende Schleife läuft so lange, bis der Mausknopf vom Benutzer gedrückt wird (`UNTIL Button`). In dieser Schleife wird durch **GetMouse** die aktuelle Mausposition abgefragt und geprüft, ob sich diese von der im Punkt **alterPunkt** gemerkten Position unterscheidet. Wenn ja, wird das Bild einmal an der alten Stelle und danach an der neuen Mausposition (in **neuerPunkt**) gezeichnet. Die neue Position des Bildes wird dann in **alterPunkt** gespeichert. Man möge mir verzeihen, daß ich die beiden Toolbox-Operationen **GetMouse** und **Button** verwende, ohne sie vorher zu erläutern. Ihre Funktion dürfte eigentlich aus ihrer Verwendung im Beispiel hervorgehen. Sie werden im Kapitel *Ereignisse* beschrieben. Trotz aller meiner Bemühungen, schrittweise vorzugehen, mußte ich in diesem Fall etwas vorgreifen, um das Beispielprogram zu ermöglichen.

Wie erreichen wir aber nun, daß sich das Bild auf dem Bildschirm scheinbar bewegt und nicht einfach bei jeder Mausbewegung zweimal auf den Bildschirm gezeichnet wird? Hierzu nutzen wir die verschiedenen Zeichenmodi von QuickDraw aus. Wie ja bereits in einem früheren Abschnitt dieses Kapitels angedeutet, eignet sich der Zeichenmodus **patXor** hervorragend dazu, eine Grafik auf dem Bildschirm darzustellen und bei Bedarf — einfach durch Zeichnen exakt derselben Grafik wieder im Modus **patXor** — wieder zu löschen.

Alle in **QDPict** gespeicherten Zeichen-Befehle zeichnen im Modus **patXor** und ermöglichen so das beschriebene Programmverhalten. Am Anfang das Programm wird das Bild einmal gezeichnet und erscheint so auf dem Bildschirm. Danach wird es jedesmal, wenn sich sich die Maus bewegt hat, einmal an der letzten Position gezeichnet und damit gelöscht und danach an der neuen Position gezeichnet, wodurch der Eindruck einer Verschiebung zustande kommt.

3.7.3 Die Initialisierung des Programms

In der Prozedur **InitProg** wird das Programm ShowQuickDraw initialisiert. Neben dem Setzen einiger Variablen auf ihre Startwerte werden hier vor allem die beiden wichtigsten Datenstrukturen des Programms angelegt. In **QDPict** wird eine Folge von QuickDraw-Befehlen in Form eines QuickDraw-Pictures aufgezeichnet, und die **clipRgn** des aktuellen GrafPorts wird zu einer Region verändert, die das Aussehen zweier sich überlappender Schlüssellöcher hat.

```
PROCEDURE InitProg;
VAR
    schluessel: RgnHandle;

    FUNCTION CreatePict: PictHandle;
    BEGIN
        ........... { wird weiter unten beschrieben }
    END; { CreatePict }


    FUNCTION CreateRgn: RgnHandle;
    BEGIN
        ........... { wird weiter unten beschrieben }
    END; { CreateRgn }
```

```
BEGIN
    InitGraf(@thePort);
    InitFonts;

    OpenPort(@port);
    SetPort(@port);

    SetPt(alterPunkt,100,100);
    SetPt(neuerPunkt,100,100);

    QDPict      := CreatePict;

    schluessel := CreateRgn;

    PenMode(patXor);
    SetRect(r,0,0,2000,2000);
    FillRect(r,dkGray);
        { GrafPort mit Grau füllen }

    FillRgn (schluessel,white);
    InsetRgn(schluessel,-3,-3);
    PenSize(2,2);
    PenPat(black);
    FrameRgn(schluessel);
        { schwarz umrandete Schlüssellöcher
          aus dem grauen Feld ausschneiden }

    InsetRgn(schluessel,3,3);

    SetClip(schluessel);
    ShowCursor;
END;{ InitProg }
```

InitProg initialisiert zunächst QuickDraw und die Zeichensatz-Verwaltung mit den beiden Aufrufen **InitGraf** und **InitFonts**. Dies ist am Anfang jedes Programmes nötig, das QuickDraw verwendet (bzw. auch Text ausgibt), wird aber in einigen Programmiersprachen vor dem Programmierer verborgen. Bevor Sie dieses Beispielprogramm selbst testen, schauen Sie also bitte einmal in der Dokumentation ihres Compilers oder Interpreters nach, ob ein Programm selbst für diese beiden Aufrufe verantwortlich ist oder ob sie "die Sprache" bereits vor dem eigentlichen Programmstart erledigt. Nachdem QuickDraw initialisiert wurde, müssen wir noch einen

GrafPort erzeugen, den wir für die folgenden Zeichenausgaben benutzen können. Hierzu dient uns die globale Variable **port**, die wir nach der Initialisierung mit **OpenPort** auch gleich als aktuellen GrafPort benennen (mittels **SetPort**).

Nach diesen "Haushaltungsaufgaben" werden die beiden Punkte **alterPunkt** und **neuerPunkt** auf plausible Startwerte gesetzt. (In der Programmier - sprache C wäre dies z.B. nicht nötig, da es dort bessere Methoden zur Initialisierung von Variablen gibt.) Danach wird in **QDPict** eine Folge von Zeichen-Befehlen gespeichert und in der Region **schluessel** ein Gebiet erzeugt, das das Aussehen zweier überlappender Schlüssellöcher hat. Sowohl das Erzeugen des QuickDraw-Pictures wie auch der Region **schluessel** delegieren wir an separate Funktionen, die weiter unten beschrieben sind.

Damit der Bildschirm ein wenig "netter" aussieht und die Begrenzung der Grafik durch die Region **schluessel** deutlicher wird, füllen wir ihn mit einem dunkelgrauen Muster und "schneiden" daraus **schluessel** aus, indem wir diese Region mit dem Muster **white** füllen. Danach bekommt **schluessel** noch einen 2 Punkte breiten schwarzen Rand.

Am Ende wird schließlich die **clipRgn** des GrafPorts mittels **SetClip** auf die Fläche **schluessel** gesetzt und mit **InitCursor** dafür gesorgt, daß der Maus-Cursor auf alle Fälle sichtbar ist und seine Normalform (einen nach links oben gerichteten Pfeil) zeigt.

3.7.4 Das Erzeugen eines QuickDraw-Pictures

Schauen wir uns nun einmal die beiden Funktionen **CreatePict** und **CreateRgn** an, die **InitProg**, das QuickDraw-Picture **QDPict** bzw. die Region **schluessel** liefern:

```
FUNCTION CreatePict: PictHandle;
VAR
    pict:         PicHandle;
    saveClip:     RgnHandle;
    r:            Rect;
BEGIN
    SetRect(r,1,1,100,100);
    saveClip := NewRgn;
    GetClip(saveClip);
    ClipRect(r);
```

```
    pict := OpenPicture(r);
        { Initialisierung des Bildes }
        PenSize(1,1);
        PenPat(black);
        PenMode(patXor);    { <-----   !!! }
        TextSize(9);
        TextMode(patXor);
        TextFont(geneva);
        FrameRect(r);       { Rahmen um das Bild }

        { Ein kleines Rechteck }
        SetRect(r,10,10,20,20);
        FrameRect(r);

        { Ein Kreisausschnitt }
        PenPat(dkGray);
        PenSize(2,2);
        SetRect(r,30,10,50,20);
        FrameArc(r,0,250);
        PenPat(ltGray);
        PaintArc(r,0,250);

        { Und eine kühne Behauptung }
        MoveTo(5,80);
        DrawString('QuickDraw ist Spitze !');
    ClosePicture;

    CreatePict := pict;


    SetClip(saveClip);
    DisposeRgn(saveClip);
END; { CreatePict }
```

In **CreatePict** sichern wir zunächst eine Kopie der aktuellen **clipRgn** des GrafPorts in einer lokalen Variablen **savePort**. Danach setzen wir die **clipRgn** auf das Rechteck, in dem wir danach das Bild zeichnen werden. Dies sollte man in der hier vorgestellten Form immer tun, sonst treten beim Zeichnen des Bildes unter Umständen die merkwürdigsten Effekte auf. QuickDraw speichert in einem Picture nämlich nicht nur eine Folge von Zeichen-Befehlen, sondern auch einige der Felder des aktuellen GrafPorts — u.a. eben die **clipRgn**.

Alle danach folgenden Zeichen-Befehle zwischen **OpenPicture** und **ClosePicture** werden gespeichert und erscheinen nicht auf dem Bildschirm. Welche Befehle dies genau sind, ist eigentlich ziemlich egal, und dem Leser steht es frei, damit herumzuexperimentieren. Für unser Beispiel ist es aber wichtig, zunächst das gesamte Rechteck, in dem das Bild gezeichnet wird, mit einem Rahmen zu versehen, damit diese Grenzen deutlich sichtbar werden.

Am Ende stellt **CreatePict** die alte **clipRgn** wieder her und liefert das gespeicherte Bild als Funktionsergebnis an **InitProg** zurück.

3.7.5 Das Erzeugen einer Region

```
FUNCTION CreateRgn: RgnHandle;
VAR
    rgn1,
    rgn2:           RgnHandle;
    r:              Rect;
BEGIN
    rgn1 := NewRgn;
    rgn2 := NewRgn;

    OpenRgn;
        SetRect(r,100,100,200,200);
        FrameOval(r);
    CloseRgn(rgn1);      { Ein Kreis }
    OpenRgn;
        MoveTo(150,100);
        LineTo(100,300);
        LineTo(200,300);
        LineTo(150,100);
    CloseRgn(rgn2);      { Ein Trapez }

    UnionRgn(rgn2,rgn1,rgn1);
        { rgn1 enthält jetzt ein Schlüsselloch }
    CopyRgn(rgn1,rgn2);
        { rgn2 enthält jetzt dasselbe  "        }
    OffsetRgn(rgn1,50,-50);
    OffsetRgn(rgn2,150,-100);
    UnionRgn(rgn2,rgn1,rgn1);
        { rgn1 enthält jetzt 2 Schlüssellöcher  }
```

```
    CreateRgn := rgn1;
    DisposeRgn(rgn2);
END; { CreateRgn }
```

CreateRgn nutzt vor allem die QuickDraw-Operationen, mit denen Regionen nach ihrer Erzeugung manipuliert werden können. Hierzu dienen zwei Variablen **rgn1** und **rgn2**. Zunächst werden **rgn1** auf eine kreisförmige und **rgn2** auf eine trapezförmige Region gesetzt. Diese beiden Regionen werden dann mit **UnionRgn** zu einer Schlüsselloch-förmigen Region verschmolzen. Von dieser stellen wir sofort eine Kopie her, so daß danach **rgn1** und **rgn2** beide dieselbe Region enthalten. Dann werden die beiden Schlüssellöcher aber mit **OffsetRgn** in unterschiedliche Richtungen verschoben und erneut zu einer Region verschmolzen. Das Endergebnis steht danach in **rgn1** und enthält zwei überlappende schlüssellochförmige Gebiete. Diese Region wird dann als Funktionsergebnis an **InitProg** zurückgegeben und die überflüssige Region **rgn2** wird mit **DisposeRgn** gelöscht.

3.7.6 Das Zeichnen eines gespeicherten Bildes

Was jetzt noch fehlt, um ShowQuickDraw zu vervollständigen, ist nur noch die Prozedur **ZeichneBild**.

```
PROCEDURE ZeichneBild(bild: PicHandle; pt: Point);
VAR
    r:  Rect;
BEGIN
    r := bild^^.picFrame;
    OffsetRect(r,-r.left,-r.top);
        { r.topLeft ist nun gleich (0,0) }
    OffsetRect(r,pt.h,pt.v-r.bottom);
        { linke untere Ecke von r liegt nun bei pt }
    DrawPicture(bild,r);
END;{ZeicheBild}
```

ZeichneBild stellt zunächst das Rechteck fest, in dem das Bild **bild** ursprünglich gezeichnet wurde, und legt eine Kopie davon in die lokale Variable **r**.. Mit zwei **OffsetRect**-Befehlen wird **r** dann so verschoben, daß seine untere linke Ecke an der Stelle **pt** liegt. In dieses Rechteck hinein wird **bild** dann mit einem **DrawPicture**-Aufruf gezeichnet.

3.8 Schlußbemerkung zu QuickDraw

Die Beschreibung von QuickDraw stellt das längste zusammenhängede Kapitel dieses Buches dar. Dies ist auch kein Wunder. QuickDraw ist derjenige Teil der ToolBox, mit dem jedes Programm zu tun hat und viele Programme auch am intensivsten zu tun haben. Jede Kommunikation mit dem Benutzer spielt sich am Ende immer über QuickDraw ab, und deshalb kann die Wichtigkeit dieses Teils der ToolBox gar nicht genug betont werden.

Erst wenn die wichtigsten Befehle von QuickDraw dem Programmierer in Fleisch und Blut übergegangen sind, kann er versuchen, die anderen Aspekte des Macintosh wirklich zu verstehen. Es kann deshalb nicht schaden, vor dem Weiterlesen noch ein wenig mit QuickDraw herumzuspielen.

Das oben vorgestellte Beispielprogramm gibt einen idealen Rahmen für ein Experimentieren mit QuickDraw ab. Das gespeicherte Bild kann z.B. beliebig modifiziert werden, so lange, bis fast jede der möglichen Grundformen von QuickDraw darin erscheint. Wenn man ein wenig mit dem Zeichenmodus des Bildes spielt (im Moment ist es **patXor**), kann man die interessantesten Effekte am Bildschirm erzielen. Durch **UnionRgn** und **SectRgn** kann die **clipRgn** nahezu beliebige Formen erhalten. Auch innerhalb des gespeicherten Bildes können Regionen verwendet werden.

Bei allem Staunen über die Ergebnisse seines Experimentierens sollte der Leser aber nie vergessen, immer wieder zu hinterfragen, warum die letzte Programmänderung gerade diese Wirkung auf dem Bildschirm erzielt hat, die er nun sieht!

4 Fenster-Verwaltung

In diesem Kapitel soll der Leser mit der Fenster-Verwaltung im ROM des Macintosh vertraut gemacht werden, dem sogenannten WindowManager. Neben der Maus sind die Bildschirm-Fenster wohl das erste, was einem Betrachter des Macintosh auffällt. Solche Fenster sind nicht von Apple für den Macintosh erfunden worden, sondern stammen, wie viele andere Macintosh-Konzepte, aus den Labors von XEROX. Die Grundidee dabei ist ungefähr die eines virtuellen Bildschirms. Dieser Bildschirm ist (nahezu) beliebig groß, und nur ein Teil davon ist gerade im Innern des Fensters zu sehen.

Ein ähnlicher Ansatz liegt vielen heutigen Textverarbeitungs-Programmen zugrunde und dürfte dem Leser deshalb wohl vertraut sein. Bei diesen Programmen ist der ganze Bildschirm (abzüglich eines gewissen Platzes für Kontroll- und Status-Informationen) ein Fenster, durch das man in ein Dokument blickt, das viele tausend Zeilen lang sein kann. Mit verschiedenen Tasten-Kombinationen kann man das Dokument hinter dem Fenster verschieben und kann somit andere Ausschnitte des Dokumentes betrachten.

4.1 Überblick über das Fensterkonzept des Macintosh

Das Fensterkonzept auf dem Macintosh geht jedoch etwas weiter. Nicht der ganze Bildschirm ist ein Fenster, mit dem man auf ein viel größeres Dokument herabblicken kann, sondern es können mehrere Fenster gleichzeitig auf dem Bildschirm liegen, die entweder Ausschnitte verschiedener Dokumente oder verschiedene Teile desselben Dokuments zeigen. Jedes Fenster besitzt eigene "Kontrollen", mit denen das Dokument, von dem es einen Teil zeigt, dahinter verschoben werden kann.

Diese Fenster können am Bildschirm auch beliebig positioniert werden und können sich teilweise überdecken, was dazu führt, daß der Inhalt (teilweise) verdeckter Fenster nur noch zum Teil sichtbar ist. Die Information, die im Fenster sichtbar ist — sowieso meist nur ein Teil eines größeren Dokuments — wird also noch weiter verringert. Durch die Überdeckung kommt es in

gewisser Weise zu einer Schichtung oder Hierarchie von Fenstern, bei denen immer vorne bzw. oben liegende die dahinter liegenden verdecken. Im allgemeinen ist dabei das oberste Fenster das aktuelle, in dem auch die Bearbeitung des dargestellten Dokuments durch den Benutzer stattfindet. Man kann ein "hinten" liegendes Fenster rasch nach "vorne" holen, indem man darauf mit der Maus klickt. Dahinter steckt die Annahme, daß ein Klick mit der Maus auf ein Fenster ein Zuwenden der Aufmerksamkeit zu diesem Fenster bedeutet.

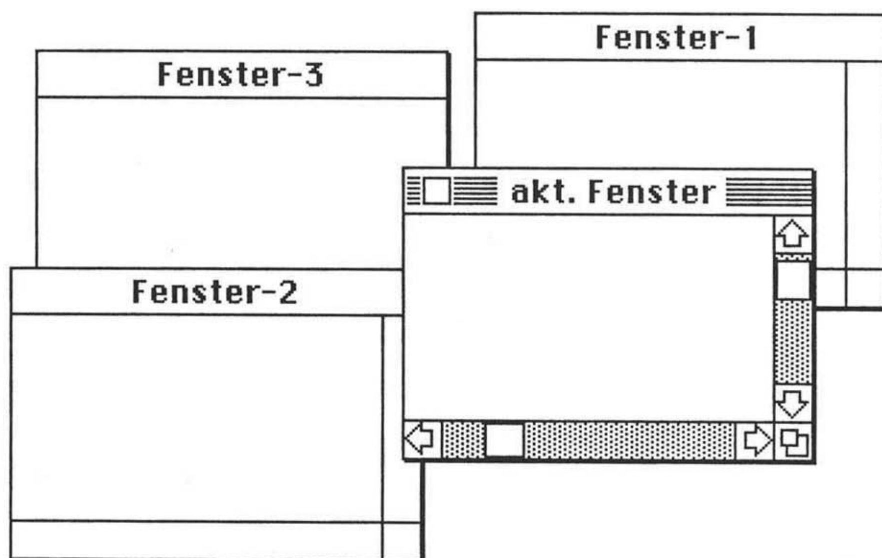


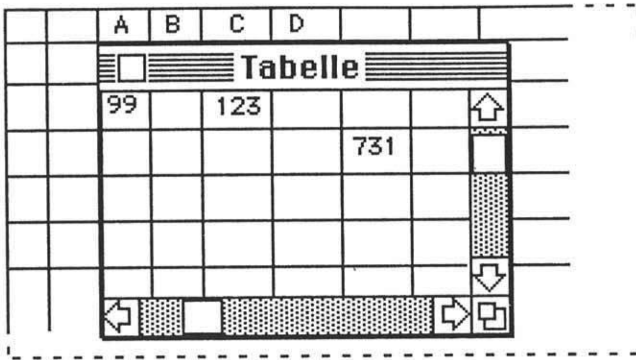
Bild 4 - 1: *Überlappende Fenster am Bildschirm*

Man könnte diese Fenster auch mit Blättern von Papier vergleichen, die auf einem Schreibtisch liegen. Man holt diese Blätter bei Bedarf hervor, um sie zu betrachten oder etwas darauf zu notieren, und verdeckt dadurch natürlich die darunterliegenden. Das wichtigste Blatt wird im allgemeinen immer auch das oberste sein. Die Analogie geht sogar noch etwas weiter. Wer normalerweise Ordnung auf seinem materiellen Schreibtisch hält, wird im allgemeinen auch seine Fenster auf dem Bildschirm geordnet haben und diejenigen schnell finden, die er benötigt. Auf wessen Schreibtisch aber Chaos herrscht, der wird auch am Bildschirm Schwierigkeiten haben, ein beiseitegelegtes Fenster wiederzufinden.

Das Fenster-Konzept, so wie es der Macintosh bietet, ist in erster Linie eine Möglichkeit, diese Arbeitsweise mit Papier auf den Computer zu übertragen und auf der begrenzten Fläche des Bildschirms mehr darzustellen als "eigentlich" darauf paßt. Das mag paradox klingen, ist aber eine der wichtigsten Funktionen von Fenstern. Indem unter jedem Fenster noch mehrere andere liegen können, die ja alle ungefähr die Größe des Bildschirms haben können, kann ein vielfaches an Information auf den Bildschirm gebracht werden. Man sieht diese Informationen zum Teil zwar nicht, da sie vielleicht gerade verdeckt sind, kann sie aber rasch sichtbar machen, sobald man sie braucht. Vor allem ist diese Art des Wechselns zwischen zwei Dokumenten in hohem Grade intuitiv und bedarf nur kurzer Lernzeit.

Ein Fenster ist allerdings noch etwas mehr als nur ein Blatt Papier. "Unter" jedem Fenster liegt eine (nahezu) beliebig große Ebene, in die mit den aus dem letzten Kapitel bekannten QuickDraw-Befehlen eine Grafik gezeichnet wird. Diese Ebene werde ich im folgenden "den View" nennen. Nur ein sehr kleiner Ausschnitt dieser großen Fläche ist innerhalb des Fensters zu sehen. Dehnt sich die Grafik über eine größere Fläche aus, als das Fenster umschließt, so existieren meist Vorrichtungen, den sichtbaren Ausschnitt zu verschieben. Ein Fenster ist also genau das, was sein Name schon sagt: ein Fenster, mit dem man auf eine meist größere Grafik "hinabschauen" kann, die "dahinter" oder "darunter" liegt.

Der View selbst enthält meist eine Abbildung von irgendwelchen Daten. Der View hinter einem Multiplan-Fenster z.B. enthält eine Darstellung eines mathematischen Modells in Form einer Tabelle. Aus den Daten, die an anderer Stelle des Programms gespeichert sind, und die nicht zwangsläufig mit dem Fenster verbunden sein müssen, läßt sich die entsprechende Abbildung jederzeit wiederherstellen. Es kann sogar sein, daß es zwei verschiedene Abbildungen für dieselben Daten gibt. Ein gutes Beispiel hierfür sind Programme wie Microsoft Chart und Lotus Jazz. Derselbe Datensatz kann einmal als Tabelle betrachtet werden und ein anderes Mal als Kuchengrafik. Beide Abbildungen können eventuell sogar zur gleichen Zeit in verschiedenen Fenstern sichtbar sein. Die beiden Fenster zeigen dann jeweils Ausschnitte aus zwei verschiedenen Views derselben Daten.



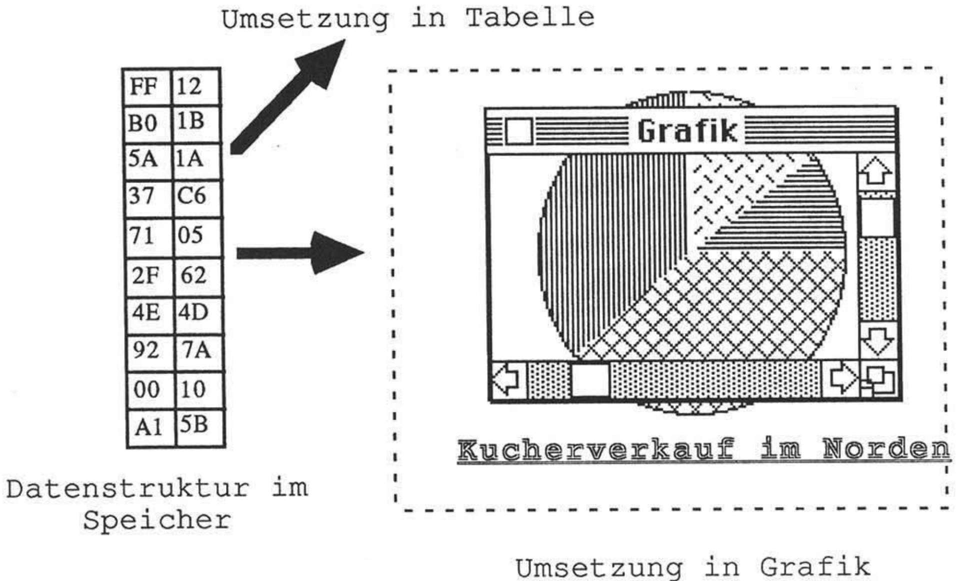


Bild 4 - 2: *Daten, Views und Fenster*

Für die meisten Teile eines Programms sind nur die Views und die Datenstrukturen, die durch sie dargestellt werden, von Interesse. Sieht man von Optimierungen zur Zeitersparnis ab, so kümmert sich ein Programm nie darum, welcher Teil des Views gerade im (in) Fenster(n) dargestellt wird. Die QuickDraw-Zeichenoperationen nutzen die gesamte Grafik-Ebene im Rechteck ((-32K,-32K) ,(32K,32K)). Welchen Teil er sich davon anschaut, ist Sache des Benutzers. Die Abbildung eines Teils dieser Ebene auf das

Innere eines Fensters übernimmt der WindowManager in Zusammenarbeit mit QuickDraw.

4.2 Anatomie eines Fensters

Ich möchte nun kurz, bevor ich auf die programmtechnische Seite von Fenstern eingehe, ihr Aussehen und die grundsätzlichen Möglichkeiten ihrer Manipulation — besonders durch den Benutzer — beschreiben. Vielen Lesern wird das Verhalten von Fenstern bereits von den Anwendungs - programmen des Macintosh her bekannt sein, ich möchte es aber trotzdem beschreiben, da kaum ein Anwendungsprogramm "alles richtig macht".

4.2.1 Das Aussehen eines Fensters am Bildschirm

Es können eine ganze Reihe von verschiedenen Fenstern mit verschiedenen Fähigkeiten auf dem Bildschirm liegen. Üblich auf dem Macintosh sind z.B. rechteckige Fenster mit und ohne Möglichkeiten zum Scrollen, mit und ohne kleinen Schatten und mit und ohne Titel-Leiste, in der der Name des Fensters erscheint. Ferner gibt es auch Fenster mit abgerundeten Ecken, und wer die Mühe nicht scheut, kann auch problemlos dreieckige oder ovale Fenster programmieren. Ich möchte hier hauptsächlich die sogenannten Dokument-Fenster behandeln, da es die gebräuchlichsten und auch komplexesten Fenster sind, die alle möglichen Eigenschaften von Fenstern besitzen. Die folgende Abbildung zeigt das Aussehen eines typischen Dokument-Fensters, wenn es an oberster Stelle liegt.

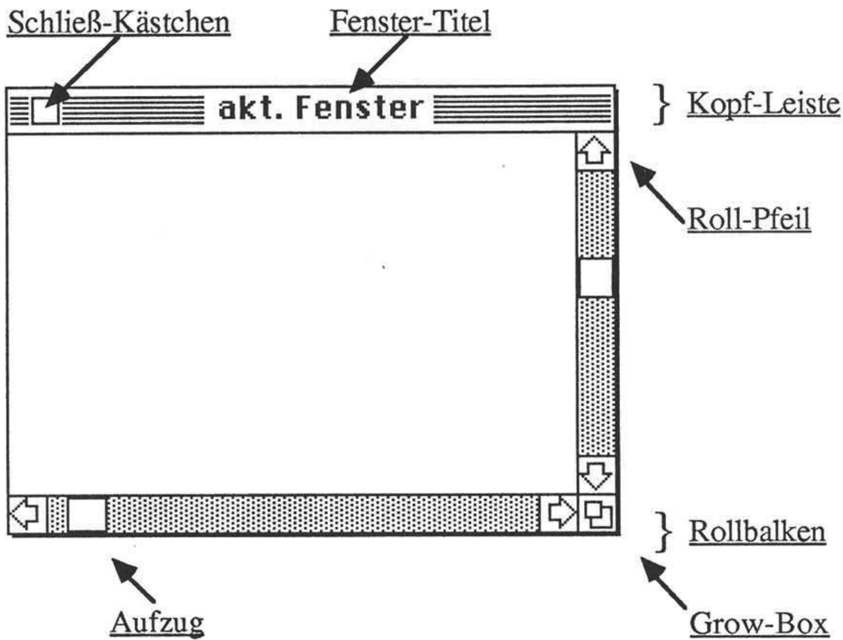


Bild 4 - 3: *"Typisches" Aussehen eines Fensters*

Die einzelnen Komponenten, aus denen sich ein Fenster zusammensetzt, zerfallen in zwei große Bereiche: den Fenster-Inhalt (engl. *content*) und den Fenster-Rahmen (engl. *frame*). Zum Fenster-Rahmen gehören zum Beispiel die Kopf-Leiste des Fensters, in der auch der Titel erscheint und der dünne schwarze Rand, sowie der Schatten unter dem Fenster. Der WindowManager ist nur zuständig für den Rahmen und somit für das Aussehen des Fensters selbst. Die Darstellung des Views erfolgt im Innern des Fensters und ist für ihn nicht mehr von Interesse. Viele der Komponenten in der obigen Abbildung sind sogenannte "Kontrollen", die dem Benutzer erlauben, das Fenster zu beeinflussen — entweder das Aussehen des Fensters selbst oder des dargestellten Dokuments. Die meisten dieser Kontrollen gehören zum Fenster-Rahmen. Die beiden Rollbalken, die es erlauben, den Ausschnitt des Dokumentes, der im Fenster sichtbar ist, zu verschieben, gehören allerdings schon nicht mehr zum Fenster-Rahmen, sondern zum Inhalt und werden auch nicht vom WindowManager verwaltet.

Die anderen Kontrollen dienen zur Beeinflussung des Fensters selbst. Jede Kontrolle ist dabei ein bestimmter Bereich, in denen Maus-Klicks – und

eventuell Bewegungen der Maus bei gedrückter Taste – Folgen für das Fenster haben. Diese Bereiche werden auch optisch besonders hervorgehoben — der Benutzer sieht sie eben als die Kontrollen in Abbildung 4 - 2. Wichtig sind aber vor allem die Bereiche, in denen die Maus diese besondere Wirkung hat. Die Bereiche sind bei einem Dokument-Fenster zwar alle rechteckig, können aber bei anderen Fenster-Typen theoretisch jedes beliebige Aussehen und jede beliebige Form haben. Es handelt sich dabei um QuickDraw-Regions, die uns ja aus dem letzten Kapitel bekannt sind.

Als Namen der verschiedenen Regionen verwende ich die englischsprachigen Bezeichnungen, wie sie auch in *Inside Macintosh* vorkommen, um diejenigen, die IM parallel lesen, nicht unnötig zu verwirren.

• Die **Structure Region** umfaßt den gesamten Bereich, den das Fenster auf dem Bildschirm einnimmt, und umschließt per Definition alle anderen Bereiche — üblicherweise aber noch etwas mehr darüber hinaus. Ein Maus-Klick irgendwo in diesem Bereich wird vom WindowManager als diesem Fenster zugehörig betrachtet.

• Die **Content Region** ist das Fenster-Innere, für das der WindowManager weitgehend nicht zuständig ist. Hier wird ein Teil des Views gezeigt, und auch Rollbalken liegen üblicherweise innerhalb der Content Region. Nur das Grow-Icon liegt normalerweise außerhalb der **Content Region** und wird vom WindowManager als solches erkannt, obwohl es im Innern des Fenster-Rechtecks liegt. Ein Maus-Klick in die **Content Region** wird vom WindowManager nicht als Betätigung einer Kontrolle betrachtet. Klickt man allerdings in die **Content Region** eines nichtaktiven Fensters, so kommt es nach vorne (wird aktiv).

• Die **Grow Region** ist der Bereich, in dem das Grow-Icon gezeichnet wird. Ein Maus-Klick in die **Grow Region** leitet im allgemeinen eine Größenveränderung des Fensters ein. Solange die Maustaste gedrückt bleibt, folgt ein graue Umrißlinie den Mausbewegungen. Deren linke obere Ecke bleibt fest bei der oberen Ecke des Fensters, und die rechte,untere Ecke wandert mit der Maus. Läßt man die Maustaste los, so ändert das Fenster seine Größe entsprechend der zuletzt sichtbaren Umrißlinie. Viele Fenster (z.B. Dialog-Fenster) besitzen kein **Grow Icon**, und ihre **Grow Region** ist leer.

• Die **Drag Region** gestattet im allgemeinen eine Bewegung des Fensters. Ein Klick in die **Drag Region** leitet eine Bewegung des Fensters ein. Bewegt man die Maus dann bei gedrückter Maustaste, so folgt eine graue Umrißlinie des Fensters der Bewegung. Läßt man die Maustaste los, bewegt

sich das Fenster zur zuletzt angezeigten Umrißlinie. Bewegt man die Umrißlinie soweit zum Bildschirmrand, daß weniger als 4 Pixel der **Grow Region** sichtbar bleiben würden, wenn das Fenster zu dieser Position bewegt würde, so erfolgt keine Veränderung des Fensters, wenn man in diesem Augenblick die Taste losläßt. Manche Fenster (z.B. die meisten Dialog-Fenster) haben keine **Drag Region** und können deshalb vom Benutzer nicht bewegt werden.

• Die **GoAway Region** dient zum Schließen. Klickt man in die **GoAway Region** und läßt die Maustaste los, solange sich der Cursor innerhalb der **GoAway Region** befindet, ist dies im allgemeinen ein Befehl zum Schließen des Fensters.

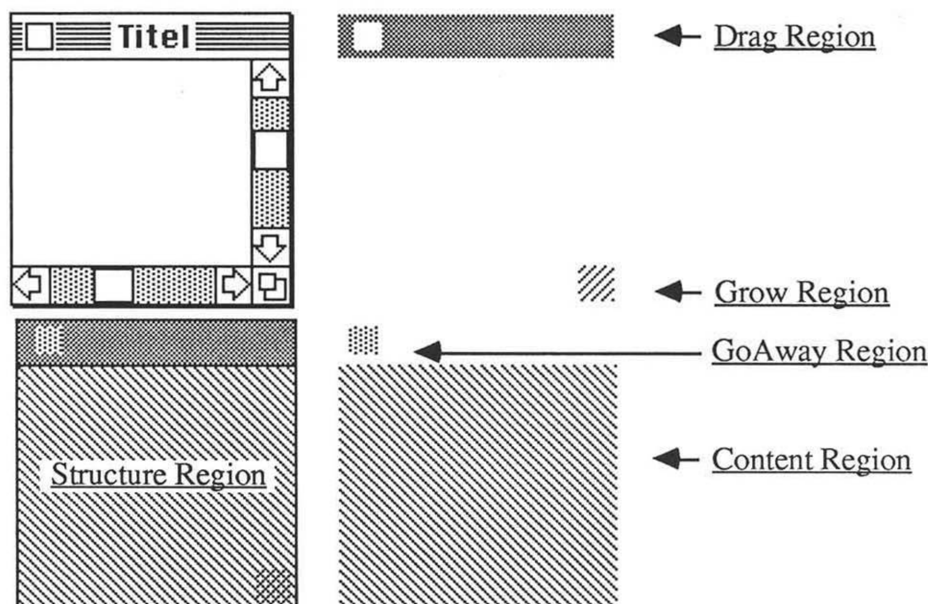


Bild 4 - 4: *Bereiche in einem Fenster*

Die meisten Regionen, die auf Maus-Klicks reagieren, existieren nur für das oberste, aktuelle Fenster. Ein Maus-Klick in die GoAway Region und die Grow Region hat die beschriebene Wirkung nur beim obersten Fenster. Die **Drag Region** allerdings existiert auch für weiter unten liegende Fenster. Ein

Maus-Klick in alle anderen Regionen außer der **Drag Region** hat bei einem hinten liegenden Fenster im allgemeinen nur die Folge, daß das Fenster nach vorne kommt.

Wenn ich in den vorangegangenen Abschnitten davon gesprochen habe, daß ein bestimmter Effekt eintritt, sobald der Benutzer in einem bestimmten Bereich eines Fensters klickt, so heißt dies jedoch nicht, daß diese Wirkung automatisch eintritt. Das gerade aktive Anwendungs-Programm muß explizit mit Funktionsaufrufen an den WindowManager dafür sorgen, daß diese Effekte eintreten. Deswegen habe ich auch immer betont, daß dieser Effekt **im allgemeinen** eintritt. Es handelt sich hierbei um Regeln, die in einem Programm erst noch verwirklicht werden müssen. Man kann auch ganz andere Regeln für die Programme, die man selbst programmiert, festlegen. Es ist aber sinnvoll, den hier beschriebenen Regeln zu folgen, um den Benutzer, der das Verhalten anderer Programme kennt, nicht zu verwirren.

Übrigens keine Angst vor dem Aufwand für die Realisierung dieser Regeln in eigenen Programmen! Der Programmier-Aufwand dafür ist nicht allzu groß. Im allgemeinen muß ein Programm, nachdem ein Maus-Klick erfolgt ist, nur feststellen, in welchen Bereich eines Fensters er fiel (auch dabei hilft der WindowManager), und dann eine Operation des WindowManagers aufrufen, die dann die oben beschriebene Aktion ausführt. Das Ganze ist denkbar einfach, wie wir in den Beipiel-Programmen sehen werden, und läßt doch noch eine Menge Möglichkeiten für den Programmierer offen, in den "normalen" Ablauf einzugreifen.

4.2.2 Fenster als Datenstruktur

Die Datenstruktur, die für die ToolBox ein Fenster definiert, besteht im wesentlichen aus einem QuickDraw-GrafPort, den wir ja ausführlich im letzten Kapitel kennengelernt haben. Fenster sind normalerweise die Hauptanwendung für GrafPorts auf dem Macintosh. Werden sie nicht von einem Programm für irgendwelche besonderen Zwecke verwendet, hängt jeder GrafPort mit genau einem Fenster zusammen. Die folgende Pascal-Definition beschreibt ein Fenster, so wie der WindowManager es sieht:

```
TYPE WindowRecord =
    RECORD
        port:           GrafPort;
        windowKind:     INTEGER;
        visible:        BOOLEAN;
        hilited:        BOOLEAN;
        goAwayFlag:     BOOLEAN;
        spareFlag:      BOOLEAN;
        strucRgn:       RgnHandle;
        strucRgn:       RgnHandle;
        contRgn:        RgnHandle;
        updateRgn:      RgnHandle;
        windowDefProc:  Handle;
        dataHandle:     Handle;
        titleHandle:    StringHandle;
        titleWidth:     INTEGER;
        controlList:    Handle;
        nextWindow:     WindowPeekl;
        windowPic:      PicHandle;
        refCon:         LongInt;
    END { WindowRecord };

    WindowPeek = ^WindowRecord;
    WindowPtr  = GrafPtr;
```

Diese Definition ist etwas "trickreich". Der Typ **WindowPeek** wird als Zeiger auf einen **WindowRecord** deklariert und **WindowPtr** gleich **GrafPtr** erklärt. Variablen, die auf Fenster verweisen, sind dabei üblicherweise vom Typ **WindowPtr**. Dies hat den Vorteil, daß man sie verwenden kann wie **GrafPtr** (Zeiger auf **GrafPorts**). Allen QuickDraw-Operationen, die einen **GrafPtr** als Parameter erwarten, kann man also genausogut einen **WindowPtr** übergeben. Dies ist deshalb nicht problematisch, da ein WindowRecord als erstes Feld ja einen **GrafPort** hat und deshalb an der Stelle im Speicher, auf die ein **WindowPtr** zeigt, ja auch ein **GrafPort** steht.

Praktisch ist diese Gleichsetzung der beiden Typen deshalb, da Fenster vom Programm aus meist wie **GrafPorts** behandelt werden und die zusätzlichen Felder der Datenstruktur **WindowRecord** nicht von Interesse sind. Werden diese Felder benötigt, so muß man eine Variable vom Typ **WindowPeek** als Zeiger auf das entsprechende Fenster deklarieren. (In Programmiersprachen

mit Typschutz, wie Pascal, muß man dazu aber den Typschutz auf mehr oder weniger umständliche Weise umgehen. Wie dies aussehen kann, dafür werden wir noch einige Beispiele in den folgenden Kapiteln kennenlernen.)

Diese zusätzlichen Felder sind auch mit Vorsicht zu genießen. Es ist gefahrlos möglich, ihre Werte zu lesen, jedoch riskant, sie zu modifizieren. Dies liegt daran, daß sie teilweise voneinander abhängen, die Änderung eines Feldes also Änderungen in anderen Feldern bedingt. Bevor einem Fenster und GrafPorts nicht sehr vertraut sind, sollte man diese Felder deshalb am besten unangetastet lassen bzw. für Änderungen die dafür vorgesehenen Prozeduren verwenden, die die gegenseitigen Abhängigkeiten berücksichtigen.

• **port** ist der **GrafPort**, dessen Grafik-Ebene (View) in diesem Fenster dargestellt wird. Er bestimmt das Koordinaten-System und alle anderen für QuickDraw relevanten Status-Informationen für dieses Fenster. **port.portRect** entspricht z.B. bei Dokument-Fenstern der **Content Region** des Fensters.

• **windowKind** ist ein Feld, in dem der Programmierer einen Wert eintragen kann, der ihm helfen kann, unter verschiedenen Fenstern zu unterscheiden. Werden in einem Programm sowohl Fenster verwendet, die Texte zeigen, wie auch solche, die Grafik zeigen, so wäre es vielleicht sinnvoll, dies durch unterschiedliche Werte in **windowKind** zu markieren. Werte zwischen 1 und 7 sind allerdings bereits für eine Verwendung durch das Betriebssystem reserviert, und negative Werte dürfen auch nicht eingetragen werden. Alle Werte ab 8 sind allerdings frei.

• wenn **visible** gleich TRUE ist, wird dieses Fenster zur Zeit am Bildschirm gezeigt. Ist **visible** FALSE, ist das Fenster im Moment unsichtbar (aber nicht geschlossen und gelöscht).

• wenn **hilited** gleich TRUE ist, so handelt es sich bei diesem Fenster um das vorderste am Bildschirm. Alle Kontrollen werden dann gezeichnet und sind wirksam.

• wenn **goAwayFlag** gleich TRUE ist, so besitzt das Fenster ein Schließ-Kästchen (bei Dokument-Fenstern in der oberen linken Ecke der Titel-Leiste) und kann über diese geschlossen werden. Viele Fenster besitzen keins, und bei diesen ist **goAwayFlag** gleich FALSE.

• **spareFlag** ist für zukünftige Erweiterungen reserviert.

• **strucRgn**, und **contRgn** sind Handles für die Regionen, die in Abschnitt 4.2.1 erläutert wurden. **updateRgn** wird weiter unten im Abschnitt 4.3.3 noch erläutert.

• **windowDefProc** ist ein Handle auf die Prozedur, die dieses Fenster definiert. Diese Prozedur wird vom WindowManager immer dann aufgerufen, wenn dieses Fenster an irgendwelchen Operationen beteiligt ist. Dieses Feld sollte nur von sehr fortgeschrittenen Anwendern benötigt werden! **dataHandle** ist für die Verwendung durch die **windowDefProc** reserviert.

• **titleHandle** ist ein Handle auf den Namen (einen Text im Pascal-String-Format) des Fensters, der im Kopf dargestellt wird, und **titleWidth** ist die Breite dieses Titels in Pixeln.

• **controlList** und **nextWindow** werden vom WindowManager intern benötigt und enthalten nur für fortgeschrittene Programmierer Informationen. Diese Felder sollten nicht geändert werden!

• **windowPic** ist ein Handle auf das Bild, das im Innern des Fensters dargestellt wird, und wird in Abschnitt 4.3.3 näher erläutert.

• das Feld **refCon** schließlich steht zur freien Verfügung des Anwenders. Hier kann man Daten oder Zeiger bzw. Handles auf Daten deponieren, die mit diesem Fenster in Verbindung stehen.

4.3 Die Aufgaben und Leistungen der Fenster-Verwaltung

Alle "wesentlichen" Probleme im Zusammenhang mit der Verwaltung von überlappenden Fenstern auf dem Bildschirm nimmt der WindowManager in Zusammenarbeit mit QuickDraw dem Programmierer ab. Diese Probleme haben vor allem mit der "Einstellung" des GrafPorts zu tun. Eine der Hauptaufgaben des GrafPorts ist es ja, wie wir im Kapitel QuickDraw gesehen haben, eine Beziehung zwischen der abstrakten Grafik-Ebene und den Objekten auf ihr, wie Punkten und Rechtecken, und den konkreten Pixeln am Bildschirm zu schaffen. Arbeitet ein Programmierer direkt mit GrafPorts, so ist er selbst für die Einstellung des GrafPorts verantwortlich. So muß er z.B. festlegen, welcher Teil des Bildschirms genutzt wird und welches Rechteck die Bits der Bildschirm-BitMap in der Grafik-Ebene einnehmen. Da dies nicht so ganz einfach ist, beläßt man es meist bei den

Voreinstellungen des GrafPorts, die für die meisten einfachen Anwendungen genügen.

Arbeitet man jedoch statt direkt mit GrafPorts mit Fenstern, wird die ganze Angelegenheit deutlich schwieriger. Das Verschieben, Vergrößern, Verkleinern und Übereinanderlegen von Fenstern funktioniert nur dann richtig, wenn dabei die GrafPorts der beteiligten Fenster entsprechend verändert werden. Diese schwierig zu koordinierenden Änderungen automatisch zu erledigen, ist — neben dem Zeichnen von Fenstern — die Hauptaufgabe des WindowManagers.

4.3.1 Koordinaten

Verschiebt man ein Fenster auf dem Bildschirm, so wandert sein Inhalt automatisch mit zur neuen Position. Der WindowManager kopiert dazu die Bits einfach von ihren alten Positionen auf die neuen Positionen. Wünschenswert ist es nun natürlich, daß mit dieser Verschiebung des Fensters keine Veränderung der Koordinaten von im Fenster dargestellten Objekten einhergeht. Haben wir z.B. vor der Verschiebung des Fensters das Rechteck ((1,2),(10,4)) gezeichnet und wollen es jetzt löschen, nachdem das Fenster verschoben wurde, so kann dies nur dann klappen, wenn die Pixel, die durch das Zeichnen des Rechtecks schwarz wurden, nach wie vor bei denselben Koordinaten liegen. Und dies, obwohl es sich ja nach der Verschiebung physikalisch um ganz andere Pixel des Bildschirms handelt. Nur logisch, im lokalen Koordinatensystem dieses Fensters bzw. GrafPorts sollten sie noch an denselben Positionen stehen.

Diese Veränderung des Koordinatensystems führt der WindowManager automatisch durch, wenn wir das Fenster nicht durch direkte Manipulation der Felder des WindowRecords verschieben, sondern mit einer der dafür vorgesehenen Prozeduren. Wenn wir innerhalb des Fensters also zweimal nacheinander ein und dasselbe Bild zeichnen, so erscheint es relativ zur linken, oberen Ecke des Fensters immer an derselben Stelle, egal wie oft und wohin das Fenster zwischendurch verschoben wurde. Der WindowManager sorgt immer dafür, daß die linke obere Ecke des Fensters dieselben Koordinaten im lokalen Koordinatensystem des Fenster-GrafPorts beibehält, wenn das Fenster verschoben wird. Nach dem ersten Öffnen des Fensters liegt die linke, obere Ecke üblicherweise bei (0,0) und bleibt auch dort, wenn das Fenster keine Rollbalken besitzt.

4.3.2 Clipping

Zeichnet man innerhalb eines Fensters eine Grafik, so möchte man natürlich, daß alle Zeichen-Befehle automatisch am Rand des Fensters haltmachen. Die einfachste Begrenzung, auf die man dabei achten muß, ist das Rechteck, das das Fenster-Innere, die **Content Region**, umschließt. Dieses Rechteck ist üblicherweise gleich dem Rechteck **port.portRect** des entsprechenden GrafPorts im WindowRecord. Kompliziert wird es aber rasch, wenn mehrere Fenster auf dem Bildschirm liegen. Wird ein Fenster teilweise von den darüberliegenden verdeckt, so kann der sichtbare Bereich die merkwürdigsten Formen annehmen. Wie man sich leicht vorstellen kann, können sich durch die Verschiebung, Vergrößerung, Verkleinerung, das Öffnen oder das Schließen nur eines Fensters die sichtbaren Regionen dieses und vieler anderer Fenster ändern. Teile werden freigelegt, andere werden neu verdeckt.

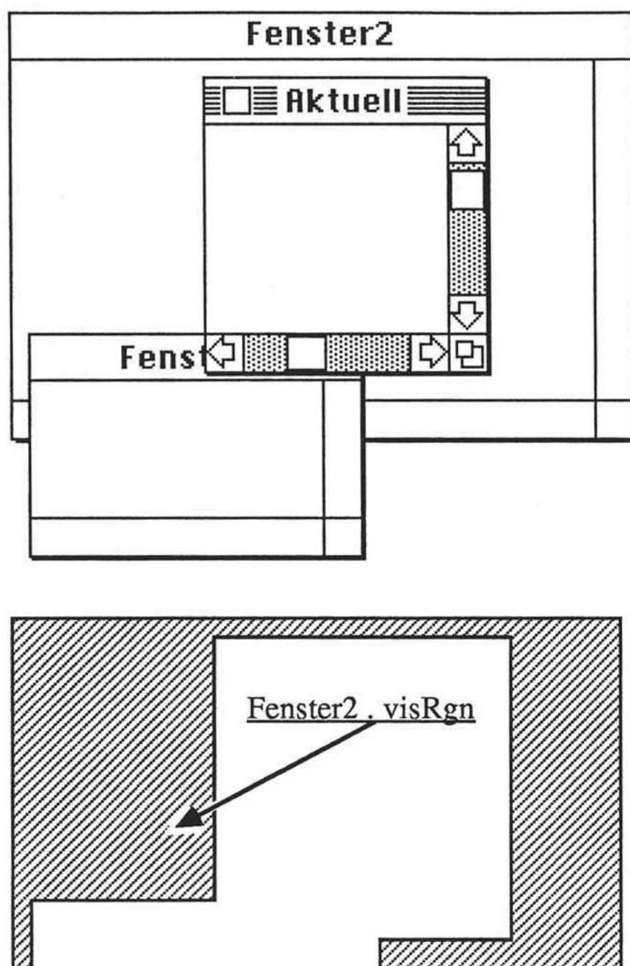


Bild 4 - 5: *Die sichtbare Region eines teilweise verdeckten Fensters*

Um diese vielfältigen Verdeckungsmöglichkeiten zu handhaben, sind natürlich die verschiedenen Begrenzungs-Regionen eines QuickDraw-GrafPorts ideal. Wie ja auch schon im letzten Kapitel erwähnt wurde, sind sie auch hauptsächlich dafür geschaffen worden. Der WindowManager verwendet vor allem die Region **visRgn** des GrafPorts. Hier ist immer diejenige Fläche eingetragen, die vom Fenster-Inhalt wirklich zu sehen ist. Um sie zu ermitteln, wird der Wert des Feldes **contRgn** des WindowRecords genommen und von dieser Region die **Structure Regions**

aller Fenster abgezogen, die logisch vor diesem Fenster liegen. Diese Berechnung führt der WindowManager immer dann aus, wenn

- ein Fenster vergrößert wurde,
- ein Fenster verkleinert wurde,
- ein Fenster verschoben wurde,
- ein Fenster vom Bildschirm verschwand,
- ein Fenster neu auf dem Bildschirm auftauchte

und zum Erreichen dieser Ereignisse die entsprechenden Prozeduren des WindowManagers verwendet wurden.

4.3.3 Auffrischen von Fenstern

Ein besonderes Problem bei sich überlappenden Fenstern ist das Auffrischen (engl. *Updating*) von freigelegten Fenstergebieten. Verschiebt man ein Fenster, so werden dadurch meist Teile anderer Fenster sichtbar, die vorher verdeckt waren. Diese Teile zeichnet der WindowManager zunächst einmal weiß. Genauso erhält ein Fenster, das frisch geöffnet wurde, zunächst einen weißen Inhalt. Wie gelangt nun die Grafik, die eigentlich im Fenster-Innern dargestellt wird, in diese weißen Flächen?

Es wäre natürlich einfach, wenn eine Kopie des gesamten Fenster-Inhalts einschließlich verdeckter Gebiete immer irgendwo abgespeichert ist und bei Bedarf einfach auf den Bildschirm geholt wird. Dies würde jedoch Unmengen an Speicherplatz kosten, da sehr viele Fenster auf dem Bildschirm liegen können. Da der WindowManager auch keine Ahnung vom logischen Aufbau der im Fenster dargestellten Grafik hat, müßte er den Fenster-Inhalt pixelweise speichern. Von Ausnahmefällen abgesehen, wird ein Fenster-Inhalt deshalb nie gespeichert, solange er nicht auf dem Bildschirm sichtbar ist. Der WindowManager verläßt sich im Gegenteil darauf, daß das gerade laufende Programm fähig ist, den kompletten Fenster-Inhalt jedes Fensters jederzeit wiederherzustellen.

Um dies zu erreichen, wird im Feld **updateRgn** des WindowRecords eine Region gepflegt, in der alle Flächen des Fenster-Inneren gesammelt werden, die aufgefrischt werden müssen, da ihr Inhalt nicht den "wirklichen" Inhalt des Views widerspiegelt. Dies sind natürlich unter anderem alle Flächen, die von anderen Fenstern verdeckt waren und freigelegt wurden und die vorübergehend mit weiß angefüllt wurden. Solange Fenster existieren, deren **updateRgn** nicht leer ist, teilt der WindowManager dies dem Programm mit, indem er ihm einen sogenannten **UpdateEvent** schickt. Dieser

UpdateEvent enthält zusätzlich auch Informationen darüber, welches Fenster aufgefrischt werden muß. Das Programm sollte auf diese Mitteilung mit dem Neuzeichnen des Fenster-Inhalts reagieren und dem WindowManager schließlich mitteilen, daß der Fenster-Inhalt aufgefrischt ist. Wie sich die Kommunikation eines Programms mit dem WindowManager genau abspielt, werden wir im Kapitel *Ereignisse* sehen.

Wichtig sind aber die Konsequenzen, die sich daraus für Macintosh-Programme ergeben. Jedes Programm muß jederzeit fähig sein, den Inhalt jedes Fensters neu zu zeichnen, sobald der WindowManager es dazu auffordert. Möchte man also z.B. eine Mitteilung an den Benutzer auf den Bildschirm schreiben, so reicht es nicht aus, einfach die entsprechenden QuickDraw-Aufrufe zu machen. Es muß immer gespeichert werden, welche Informationen an welcher Stelle im Fenster stehen, um diese Aufrufe jederzeit wiederholen zu können.

Zu jedem Fenster sollte es also im Programm eine Datenstruktur geben, die das Aussehen des Views, der in diesem Fenster dargestellt wird, hundertprozentig beschreibt. Um Fehler-Möglichkeiten und Konflikte gering zu halten, sollte die Umsetzung dieser Datenstruktur in eine Grafik möglichst zentral gehalten werden. Das heißt: Es ist gefährlich und schwierig zu koordinieren, wenn alle möglichen Programmteile etwas in ein Fenster zeichnen. Viel besser läßt sich die Koordinierung verschiedener Programmausgaben regeln, wenn nur eine Prozedur die Darstellung einer zentralen Datenstruktur im View übernimmt und alle anderen Programmteile nur diese Datenstruktur modifizieren. Nur so kann sichergestellt werden, daß ein Auffrischen des Fenster-Inhalts jederzeit möglich ist.

Der WindowManager kann auch zur Kommunikation zwischen den verschiedenen Programmteilen genutzt werden. Es ist ja nicht nur dann nötig, den Fenster-Inhalt aufzufrischen, wenn Teile davon freigelegt werden. Auch wenn sich die zugrundeliegenden Daten ändern (z.B. durch Eingaben des Benutzers), muß die Darstellung dieser Daten den veränderten Gegebenheiten angepaßt werden. Hierzu wird jedesmal, wenn ein Programmteil die Datenstruktur geändert hat, die im View dargestellt wird, die Fläche in der Grafik-Ebene, die davon betroffen ist, zur **updateRgn** hinzugefügt. Der WindowManager sorgt dann dafür, daß diese Fläche neu gezeichnet wird, sofern sie (oder Teile von ihr) im Moment sichtbar ist. Hierzu dienen die **Inval**...-Aufrufe, die im folgenden Abschnitt erläutert sind.

Am einfachsten kann man sich das Auffrischen von Fenster-Inhalten mit QuickDraw-Pictures (aus mehreren Einzel-Aufrufen zusammengesetzten Bildern) machen. Hierzu dient das Feld **windowPic** des WindowRecords.

Installiert man hier eine Datenstruktur vom Typ **PicHandle**, so wird das entsprechende Bild automatisch immer dann gezeichnet, sobald das Fenster aufgefrischt werden muß (der Typ **PicHandle** sollte aus dem Kapitel QuickDraw bekannt sein). Statt den View immer dann zu zeichnen, wenn dem Programm ein **UpdateEvent** mitgeteilt wird, kann man also auch die komplette View-Grafik in einem QuickDraw-Picture speichern und braucht sich dann um ein Auffrischen des Fensters überhaupt nicht mehr kümmern.

Diese Lösung beansprucht aber sehr viel Speicherplatz, sobald die Pictures kompliziert werden, und bringt kaum Vorteile, wenn sich das Aussehen des Views häufig ändert. Sie ist nur in seltenen Fällen und auch wohl nur auf dem Macintosh mit 512K sinnvoll.

4.4 Operationen der Fenster-Verwaltung

Kommen wir nun zu den Operationen der Fenster-Verwaltung, die einem Programm zugänglich sind. Die meisten davon werden nur in der sogenannten MainEventLoop (Haupt-Ereignis-Schleife) des Programms benötigt und können deshalb auch erst an einem sinnvollen Beispiel vorgeführt werden, wenn das Ereignis-Konzept des Macintosh vorgestellt wurde.

```
PROCEDURE InitWindows;
```

Diese Prozedur dient zur Initialisierung des WindowManagers direkt nach dem Start des Programms. Sie löscht alle zu diesem Zeitpunkt auf dem Bildschirm befindlichen Fenster und reserviert Speicherplatz für die intern vom WindowManager benötigten Variablen.

```
FUNCTION NewWindow(
            wStorage:     Ptr;
            boundsRect:   Rect;
            title:        Str255;
            visible:      BOOLEAN;
            procID:       INTEGER;
            behind:       WindowPtr;
            goAwayFlag:   BOOLEAN;
            refCon:       LONGINT
        ):WindowPtr;
```

NewWindow dient zum Erzeugen (und Öffnen) neuer Fenster. Diese Funktion liefert als Wert einen **WindowPtr** auf das neuangelegte Fenster zurück. Die Parameter bestimmen vor allem das Erscheinungsbild des Fensters:

Im Parameter **wStorage** kann man **NewWindow** einen Zeiger auf eine Variable vom Typ WindowRecord übergeben, der die Datenstruktur des Fensters aufnimmt. Übergibt man aber NIL, so reserviert sich die Funktion selbst dynamisch den entsprechenden Platz für den WindowRecord im Heap. Da es sich dabei um einen nicht verschiebbaren Speicherblock handelt, sollte man wenn irgend möglich für **wStorage** eine Variable vom Typ WindowRecord reservieren oder aber **NewWindow** so früh wie möglich aufrufen, solange der Heap noch weitgehend ungenutzt ist.

BoundsRect gibt in globalen Koordinaten das Rechteck an, das das Innere des Fensters auf dem Bildschirm einnehmen soll.

Title ist die Überschrift im Kopf des Fensters, sofern es einen hat.

Visible gibt an, ob das Fenster sofort nach dem Anlegen und der Initialisierung des WindowRecords sichtbar werden soll. Ist **visible** gleich FALSE, bleibt es zunächst unsichtbar und wird erst nach einem expliziten Prozedur-Aufruf (**ShowWindow**) sichtbar.

ProcID gibt die Art und Form des gewünschten Fensters an. Einige sind vordefiniert. Eigene Fenster-Typen können vom Programmierer mit entsprechendem Aufwand geschaffen werden. Die vordefinierten Fenster-Typen können durch Übergabe der folgenden 6 Konstanten für den Parameter **ProcID** erzeugt werden:

```
CONST
    documentProc=   0;
    dBoxProx=       1;
    plainDBox=      2;
    altDBox=        3;
    nowGrowDocProc= 4;
    rDocProc=       16;
```

Die folgende Abbildung illustriert die verschiedenen Fenster-Typen, so wie sie am Bildschirm erscheinen würden:

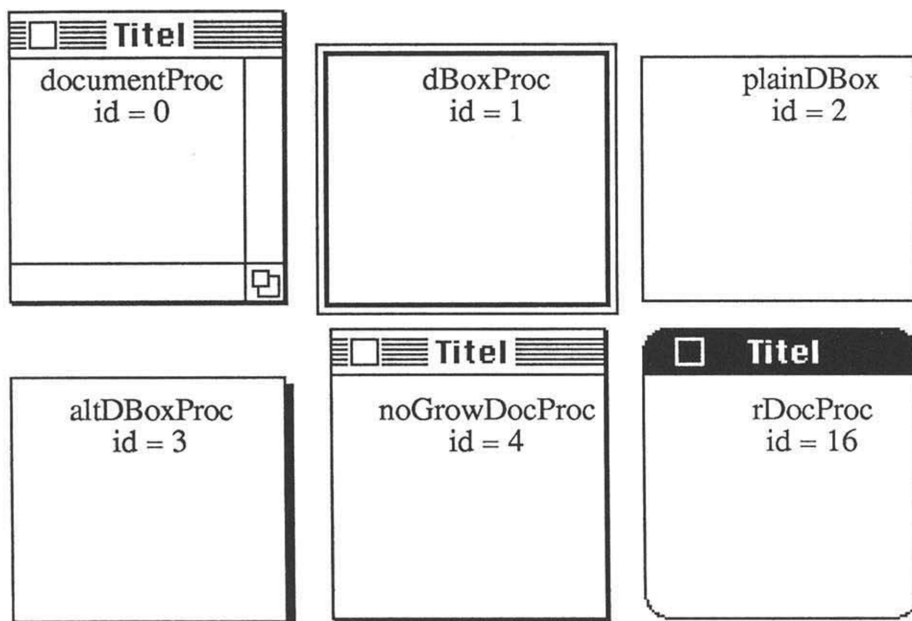


Bild 4 - 6: *Die vordefinierten Fenster-Typen*

Man achte bei der obenstehenden Abbildung darauf, daß das **boundsRect** aller 6 Fenster gleich ist, obwohl sie — durch die verschiedenen Schatten und Köpfe — sehr unterschiedliche Flächen auf dem Bildschirm einnehmen.

Behind gibt das Fenster an, hinter dem das neue Fenster erscheinen soll. Es bestimmt die Ebene in der es liegt, wenn man sich die Fenster auf dem

Bildschirm als übereinandergelegte Blätter vorstellt. Ist **behind** gleich NIL, so erscheint das neue Fenster als hinterstes hinter allen anderen. Übergibt man statt dessen -1 (in LisaPascal "POINTER(-1)"; in der Programmier - sprache C "-1L"), so taucht das neue Fenster vor allen anderen auf – der wohl üblichste Fall.

GoAwayFlag entspricht dem Feld **goAwayFlag** des WindowRecords und gibt bei einigen Fenster-Typen an, ob in der Kopf-Zeile ein Knopf auftaucht, mit dem das Fenster geschlossen werden kann.

RefCon entspricht dem Feld **RefCon** des WindowRecords und steht zur freien Verfügung des Programmierers.

```
FUNCTION GetNewWindow( windowID:   INTEGER;
                       wStorage:   Ptr;
                         behind:   WindowPtr
                     ):WindowPtr;
```

GetNewWindow entspricht in fast allen Punkten der Funktion **NewWindow**. Nur entnimmt der WindowManager die hier fehlenden Parameter der "Resource" vom Typ 'WIND' mit der Nummer **windowID**. Das Resource-Konzept des Macintosh wird in einem folgenden Kapitel näher beschrieben.

```
PROCEDURE CloseWindow(theWindow: WindowPtr);
```

```
PROCEDURE DisposeWindow(theWindow: WindowPtr);
```

CloseWindow und **DisposeWindow** dienen zum Entfernen eines Fensters vom Bildschirm und der Freigabe der mit dem Fenster verbundenen Datenstrukturen. **CloseWindow** ruft man auf, wenn man beim entsprechenden **NewWindow** oder **GetNewWindow** für **wStorage** einen Zeiger auf eine Variable vom Typ WindowRecord übergeben hat und **DisposeWindow**, wenn **wStorage** gleich NIL war. **DisposeWindow** gibt den, beim **NewWindow** angelegten, Speicherblock im Heap wieder frei. Ruft man **CloseWindow** auf, wenn **DisposeWindow** angebracht ist, bedeutet das Speicherverschwendung. Ruft man **DisposeWindow** auf, wenn **CloseWindow** angebracht ist, bedeutet das fast immer eine Bombe.

```
PROCEDURE  SetWTitle(    theWindow: WindowPtr;
                         title: Str255);
```

SetWTitle ändert den Namen des Fensters, auf das theWindow zeigt. Dies hat natürlich nur dann eine Wirkung, wenn Fenster des entsprechenden Typs überhaupt eine Kopfzeile mit einem Namen haben. Man sollte den Namen eines Fensters allerdings nie ohne triftigen Grund ändern! Solche Aktionen können den Anwender nur verwirren.

```
PROCEDURE  GetWTitle(    theWindow: WindowPtr;
                         VAR title: Str255);
```

GetWTitle setzt title auf den Namen des Fensters, auf das **theWindow** zeigt. Dies ist natürlich nur dann relevant, wenn Fenster des entsprechenden Typs überhaupt eine Kopfzeile mit einem Namen haben.

```
FUNCTION FrontWindow:WindowPtr;
```

FrontWindow dient dazu, einen Zeiger auf das oberste (aktive) Fenster am Bildschirm zu erhalten. Diese Funktion wird üblicherweise in Vergleichen innerhalb eines Programms benutzt, um bestimmte Programmfunktionen vom gerade oben liegenden Fenster abhängig zu machen.

```
PROCEDURE  SelectWindow(theWindow: WindowPtr);
```

SelectWindow holt theWindow "nach vorne". Danach ist **theWindow** das "aktive Fenster", das durch **FrontWindow** abgeprüft werden kann. Bei einem Dokument-Fenster erscheinen dadurch z.B. das **Grow-Icon** und die Rollbalken, die unsichtbar sind, solange das Fenster "im Hintergrund" liegt. Dies ist mit einer ganzen Reihe von Ereignissen verbunden, die dem Programm mitgeteilt werden. Darüber aber mehr im Kapitel über den EventManager. Ein Programm sollte normalerweise dann **SelectWindow** aufrufen, wenn ein Maus-Klick in das Innere dieses Fenster, getroffen hat.

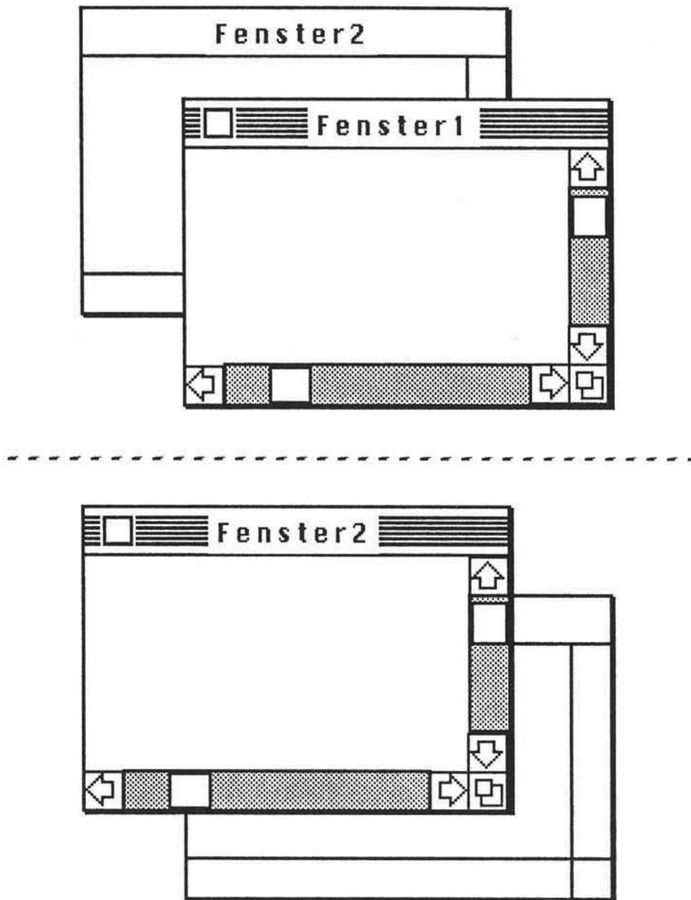


Bild 4 - 7: *Selektieren eines Fensters*

```
PROCEDURE HideWindow(theWindow: WindowPtr);
```

HideWindow läßt **theWindow** unsichtbar werden. Falls **theWindow** vorher das aktive Fenster war, so wird das nächste Fenster in der Liste aller Fenster nun aktiv. **HideWindow** verhindert aber nur, daß das Fenster am Bildschirm gezeichnet wird. Diese Prozedur löscht ansonsten weder den WindowRecord noch andere mit dem Fenster verbundene Datenstrukturen. Nach einem Aufruf von **HideWindow** ist das Feld **visible** des

entsprechenden WindowRecord gleich FALSE. War das Fenster schon vorher unsichtbar, passiert überhaupt nichts.

```
PROCEDURE ShowWindow(theWindow: WindowPtr);
```

ShowWindow ist die Umkehrung von **HideWindow**. Unsichtbare Fenster werden dadurch sichtbar. Nach einem Aufruf von **ShowWindow** ist das Feld **visible** des entsprechenden WindowRecords gleich TRUE. **ShowWindow** hebt allerdings den Effekt eines HideWindow nicht immer vollständig auf. ShowWindow verändert nämlich nicht das aktive Fenster. Ist **theWindow** das aktive Fenster, so ist es dies nach den beiden Aufrufen "**HideWindow(theWindow); ShowWindow(theWindow);**" im allgemeinen nicht mehr.

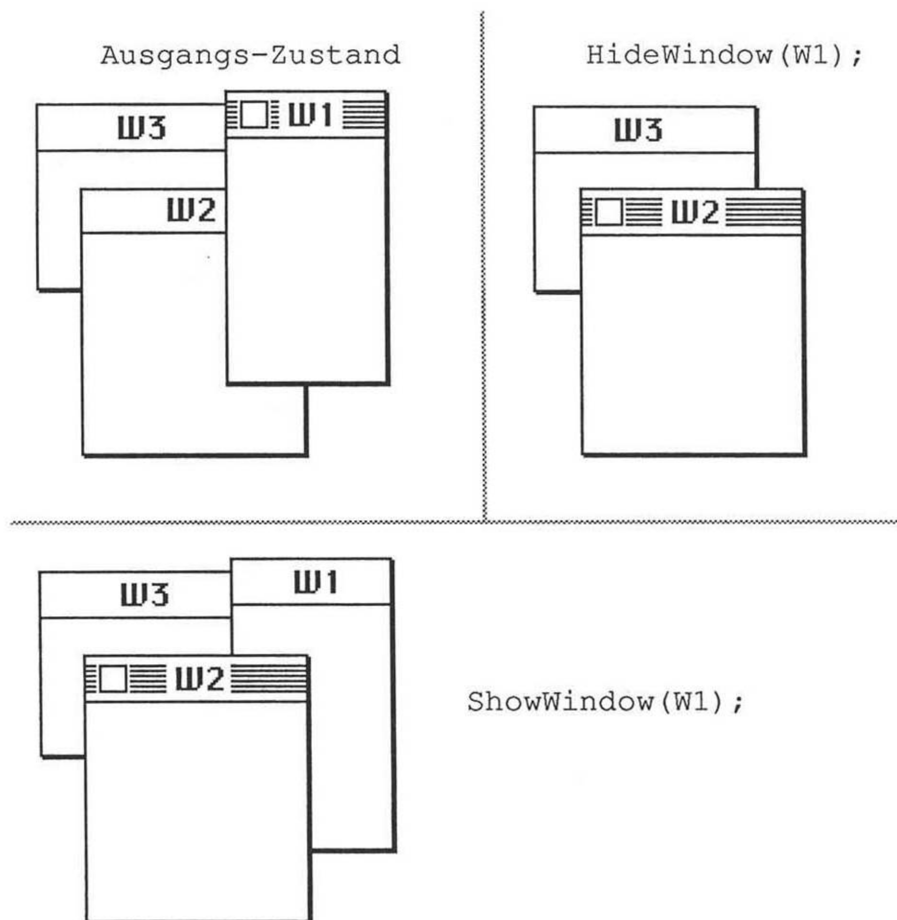


Bild 4 - 8: *Verbergen und Zeigen eines Fensters*

```
FUNCTION FindWindow(thePt: Point;
                    VAR whichWindow: WindowPtr
                   ):INTEGER;
```

FindWindow dient dazu, nach einem Maus-Klick festzustellen, was dieser getroffen hat. Dies läßt sich aus dem Funktionswert schließen, für den die folgenden Konstanten vordefiniert sind:

```
CONST
     inDesk=          0;{im grauen Hintergrund}
     inMenuBar=       1;{in der Menü-Leiste}
     inSysWindow=     2;{in einem System-Fenster}
     inContent=       3;{im Innern eines Fensters}
     inDrag=          4;{im Kopf eines Fensters}
     inGrow=          5;{im Grow Icon eines Fensters}
     inGoAway=        6;{im Close Button eines Fensters}
```

Landet der Maus-Klick in einem Fenster (Werte 1 bis 6), so enthält der VAR-Parameter **whichWindow** einen Zeiger auf dieses Fenster. Entsprechend dem Ergebnis von **FindWindow** müssen dann die entsprechenden Aktionen unternommen werden, wie sie in Abschnitt 4.1 beschrieben wurden. Wie das normalerweise im Programm aussieht, darauf werde ich im folgenden Kapitel *Ereignisse* noch näher eingehen. Im allgemeinen wird aber abhängig vom **FindWindow**-Ergebnis eine der nun folgenden Prozeduren des WindowManagers aufgerufen.

```
FUNCTION TrackGoAway(theWindow:    WindowPtr;
                     thePt:        Point
                     ):BOOLEAN;
```

TrackGoAway wird normalerweise aufgerufen, nachdem ein Maus-Klick im Schließ-Kästchen (in der **GoAway Region**) des aktiven Fensters stattgefunden hat. Die Funktion behält solange die Kontrolle, bis die Maustaste gelöst wird. Ist der Cursor dann noch innerhalb der **GoAway Region**, dann liefert sie TRUE, sonst FALSE zurück. Liefert sie TRUE, sollte **theWindow** mit **CloseWindow**, **DisposeWindow** oder **HideWindow** vom Bildschirm entfernt werden. Welcher Aufruf genau angebracht ist, hängt vom konkreten Programm ab. **TrackGoAway** hebt die **GoAway Region** auch noch in besonderer Weise hervor, solange sich der Cursor bei gedrückter Taste darin befindet. Dem Benutzer wird dadurch signalisiert, daß ein Lösen der Taste in diesem Augenblick mit dem Schließen des Fensters verbunden ist.

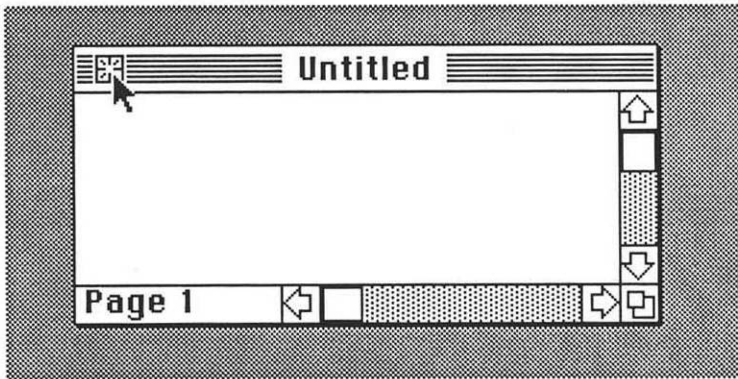


Bild 4 - 9: *Schließen eines Fensters mit der Maus*

```
PROCEDURE DragWindow(theWindow: WindowPtr;
          startPt: Point; boundsRect: Rect);
```

DragWindow wird normalerweise aufgerufen, nachdem ein Maus-Klick im Kopf eines Fensters stattgefunden hat. Die Funktion behält solange die Kontrolle, bis die Maustaste gelöst wird. Befindet sich der Cursor innerhalb von **boundsRect**, wenn die Taste gelöst wird, so wird das Fenster zu der so angegebenen Stelle am Bildschirm bewegt (mittels **MoveWindow** s.u.). Befindet sich der Cursor in diesem Augenblick außerhalb von **boundsRect**, so geschieht nichts. **DragWindow** überprüft sofort nach dem Aufruf, ob in diesem Moment die Befehlstaste gedrückt ist. Ist das der Fall, verbleibt das Fenster nach dem Lösen der Maustaste in der Ebene, in der es war. Ist die Befehlstaste nicht gedrückt, wird das Fenster nach Beendigung der Bewegung mittels **SelectWindow** nach vorne gebracht.

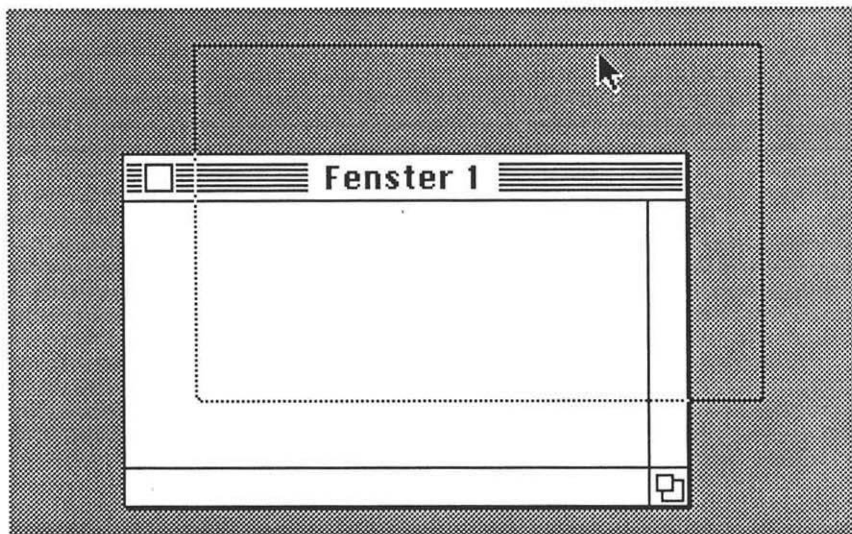


Bild 4 - 10: *Verschieben eines Fensters mit der Maus*

```
PROCEDURE MoveWindow(theWindow: WindowPtr;
                     hGlobal,
                     vGlobal:   INTEGER;
                     front:     BOOLEAN);
```

MoveWindow bewegt die obere linke Ecke des Fenster-Inneren zu den in **hGlobal** und **vGlobal** angegebenen globalen Koordinaten. Ist **front** gleich TRUE, so wird das Fenster danach mit **SelectWindow** nach vorne geholt. **MoveWindow** ändert das Koordinatensystem des entsprechenden Fenster-GrafPorts, so daß nach Beendigung der Bewegung die obere linke Ecke des Fenster-Innern nach wie vor an derselben Position im lokalen Koordinatensystem liegt wie vor der Bewegung. **MoveWindow** wird von **DragWindow** aufgerufen und leistet die eigentliche Arbeit, wenn die Bedingungen für ein Bewegen des Fensters zutreffen.

```
FUNCTION GrowWindow(theWindow:     WindowPtr;
                    startPt:       Point;
                    sizeRect:      Rect
                    ): LONGINT;
```

GrowWindow wird normalerweise nach einem Maus-Klick in der **Grow Region** des aktiven Fensters aufgerufen. Sie behält solange die Kontrolle, wie die Maustaste gedrückt bleibt, und folgt den Bewegungen der Maus mit einer grauen Umrißlinie, die die Größe des Fensters andeutet, die es erhalten würde, wenn die Maustaste in diesem Augenblick gelöst würde. **StartPt** enthält die globalen Maus-Koordinaten des Maus-Klicks und **sizeRect** schränkt die mögliche Fenster-Größe ein.

SizeRect.left enthält das Minimun der Fensterbreite,
SizeRect.top das Minimum der Fensterhöhe,
SizeRect.right das Maximum für die Fensterbreite und
SizeRect.bottom das Maximun für die Fensterhöhe.

Die Größe, die das Fenster nach dem Lösen der Maus-Taste wirklich haben soll, wird als Funktionswert zurückgeliefert. Die oberen 16 Bit des 32 Bit-Wertes enthalten die neue Fensterhöhe und die unteren 16 Bit die neue Breite. Diese 16-Bit-Werte lassen sich mit den beiden Prozeduren **LoWord** und **HiWord** aus der ToolBox ermitteln. **LoWord(GrowWindow(...))** liefert die Breite und **HiWord(GrowWindow(...))** die neue Höhe.

GrowWindow selbst verändert die Fenstergröße nicht. Dies muß das Programm selbst mit der Prozedur **SizeWindow** tun.

```
PROCEDURE SizeWindow(theWindow: WindowPtr;
          w,h: INTEGER; fUpdate: BOOLEAN);
```

SizeWindow sollte nur in Folge von **GrowWindow** aufgerufen werden. Alles andere würde den Benutzer nur verwirren. Es dient dazu, die Fenstergröße wirklich zu verändern, nachdem **GrowWindow** festgestellt hat, welche Fenstergröße der Benutzer wünscht. Das typische Programmstück dazu sieht in etwa aus:

```
erg := GrowWindow(theWindow......);
SizeWindow(theWindow,LoWord(erg),HiWord(erg),TRUE);
```

FUpdate ist ein Flag, das dem WindowManager mitteilt, ob nach der Größenveränderung ein Teil des Fenster-Inhalts neu gezeichnet (*upgedatet*)

werden soll. Dies ist bei einem Wachsen des Fensters üblicherweise nötig, da die neu hinzugekommene Fläche zunächst einmal leer ist.

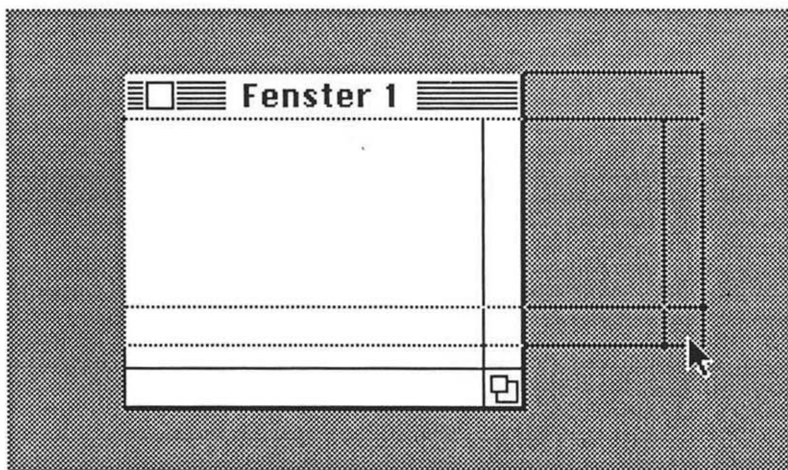


Bild 4 - 11: *Verändern der Fenstergröße mit der Maus*

```
PROCEDURE InvalRect(badRect: Rect);
```

InvalRect geht davon aus, daß der aktuelle GrafPort ein Fenster ist, und fügt zur **updateRgn** des entsprechenden WindowRecords das Rechteck **badRect** hinzu. Der Inhalt dieses Rechtecks wird auf diese Weise als "ungültig" markiert und wird beim nächsten Auffrischen des Fensters neu gezeichnet. Eine solche Markierung ist immer dann sinnvoll, wenn sich die Daten, die im Fenster dargestellt werden, verändert haben. Wie ich schon oben erwähnt habe, ist es besser, diese Veränderung nicht sofort durch direktes Zeichnen in das Fenster widerzuspiegeln, sondern dies einer zentralen Prozedur zu überlassen, die die im Fenster dargestellte Datenstruktur zeichnen kann. **InvalRect** ist die wichtigste Möglichkeit dieser zentralen Prozedur — auf dem Umweg über den WindowManager — mitzuteilen, welche Teile der Grafik neu gezeichnet werden müssen.

Ein weiterer Vorteil, Änderungen nicht sofort zu zeichnen, sondern nur mit **InvalRect** zu markieren, ist, daß so mehrere Änderungen gesammelt werden können (durch mehrere **InvalRect**-Aufrufe) und dann auf einmal gezeichnet werden, wenn die zentrale Update-Prozedur das nächste Mal aufgerufen wird. Dies spart Zeit und dem Benutzer unangenehmes Bildschirm-Flackern.

Man kann auch völlig sorglos irgendwelche Bereiche der Grafik-Ebene mit **InvalRect** markieren, ohne sich darum zu kümmern, ob sie überhaupt sichtbar sind. Der WindowManager sorgt dafür, daß das Fenster nur dann aufgefrischt wird, wenn zumindest ein Teil der "invalidierten" Fläche sichtbar ist.

```
PROCEDURE InvalRgn(badRgn: RgnHandle);
```

InvalRgn leistet genau dasselbe wie **InvalRect**, nur daß hiermit eine beliebige Region invalidiert werden kann, falls dies nötig sein sollte.

```
PROCEDURE ValidRect(goodRect: Rect);
PROCEDURE ValidRgn (goodRgn: RgnHandle);
```

ValidRect und **ValidRgn** sind die Umkehrungen von **InvalRect** und **InvalRgn**. Sie ziehen die Region **goodRgn** bzw. das Rechteck **goodRect** wieder von der **updateRgn** des entsprechenden WindowRecords ab. Dies kann sinnvoll sein, wenn sehr komplexe Objekte in einem Fenster dargestellt werden. Sobald ein Teil davon gezeichnet ist, "validiert" man die entsprechende Region und prüft dann, ob die **updateRgn** nicht schon leer ist. Wenn ja, bricht man das Zeichnen sofort ab, denn der Fensterinhalt ist dann bereits korrekt. Im allgemeinen werden diese beiden Prozeduren aber nur selten benötigt.

```
PROCEDURE BeginUpdate(theWindow: WindowPtr);
PROCEDURE EndUpdate  (theWindow: WindowPtr);
```

Mit **BeginUpdate** und **EndUpdate** klammert man üblicherweise das Zeichnen des Fensterinhaltes ein. **BeginUpdate** setzt die **visRgn** des Fenster-GrafPorts auf die Schnittfläche der alten **visRgn** und der **updateRg** des WindowRecords. Dadurch werden die Zeichenoperationen, mit denen der Fensterinhalt danach aufgefrischt wird, auf diese neue Region (und natürlich die aktuelle **clipRgn**) begrenzt. Sie werden dadurch beschleunigt, und gleichzeitig wird das Bildschirmflackern auf ein Minimum begrenzt. Typischerweise sieht der entsprechende Programmteil aus wie folgt:

```
BeginUpdate(einFenster);
  { Diverse Operationen zum Zeichnen des Inhalts };
EndUpdate  (einFenster);
```

EndUpdate stellt die alte **visRgn** des Fenster-GrafPorts wieder her und setzt die **updateRgn** auf die leere Region. Dies ist praktisch eine Mitteilung an den WindowManager, daß der Fenster-Inhalt im Moment uptodate ist.

Damit soll zunächst einmal das Kapitel "Fenster-Verwaltung" abgeschlossen sein. Die meisten der hier aufgeführten Prozeduren werden uns wiederbegegnen, sobald die Aufgaben des EventManagers besprochen werden und das erste "maclike" Programm entsteht.

5 Menüs

Eine der wichtigsten Eigenschaften des Macintosh neben den Fenstern und der Maus sind die PullDown-Menüs. Es ist seit langem erwiesen, daß die Auswahl eines Befehls aus einem Menü viel leichter zu erlernen ist als das Memorieren eines entsprechenden einzutippenden Befehls oder Control-Codes. Deshalb war es nur natürlich, einen Computer wie den Macintosh, der leicht zu bedienen und vor allem leicht zu erlernen sein sollte, mit Menüs auszustatten. PullDown-Menüs haben zudem den Vorteil, daß sie nicht viel Platz benötigen, solange man sie nicht braucht, und den Bildschirm frei - lassen, für das, wofür er eigentlich gedacht ist — Text und Grafik z.B. — die gerade bearbeitet werden.

Für den Programmierer haben Menüs noch andere Vorteile, sie sind leicht zu programmieren, wenn er die dafür vorgesehenen Funktionen der ToolBox nutzt. Sie können leicht erzeugt und verändert werden, und obwohl sie für den Benutzer sehr komfortabel ist, ist die Auswahl eines Punktes aus einem Menü von seiten des Programms völlig problemlos.

5.1 Was darf es sein? (Menüs aus der Sicht des Anwenders)

Solange der Anwender keine Auswahl aus einem Menü vornimmt, erscheinen die PullDown-Menüs des Macintosh dem Benutzer nur als eine Zeile am oberen Bildschirmrand, in dem eine Reihe von Worten schwarz (evtl. auch grau) auf weißem Grund erscheint. Diese Worte sind die "Menü-Titel" der einzelnen Menüs, und die Zeile nennt man die "Menü-Leiste". Klickt man nun mit der Maus auf eines dieser Worte, so klappt darunter ein weißer Streifen mit einem schmalen Schatten hervor, in dem normalerweise untereinander eine Reihe von Worten oder kurzen Sätzen stehen, die entweder schwarz oder grau auf dem weißen Hintergrund erscheinen. Diese Zeilen innerhalb des heruntergeklappten Streifens sind die "Menü-Punkte", unter denen man einen auswählen kann. Fährt man mit gedrückter Maus-Taste innerhalb des Streifens auf und ab, so werden die Menü-Punkte, über

denen sich die Maus gerade befindet, invertiert (sie erscheinen weiß innerhalb eines schwarzen Rahmens). Läßt man die Maus-Taste los, solange ein Menü-Punkt invertiert ist, so gilt dieser als ausgewählt, und das Programm führt eine entsprechende Aktion aus.

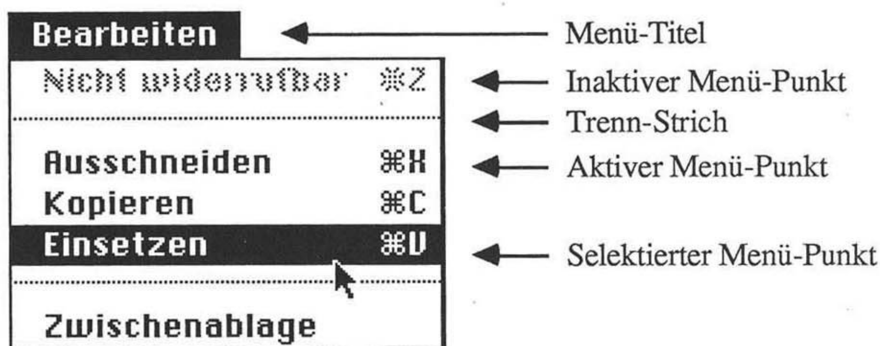


Bild 5 - 1: *Das typische Aussehen von PullDown-Menüs*

Menüs können auch ein komplizierteres Aussehen haben als gerade beschrieben. Die Menüpunkte können z.B. in verschiedenen Text-Stilen (z.B. fett oder kursiv) erscheinen und mit einem kleinen Bild versehen sein. Sie können auch überhaupt keinen Text enthalten und können auch aus einem Bild, einem Strich oder einem Muster bestehen. Sie müssen auch nicht untereinander liegen, sondern können innerhalb des weißen Streifens beliebig angeordnet sein. "Normalerweise" sieht ein Menü aber aus wie oben beschrieben.

Menü-Titel und/oder Menü-Punkte können auch grau statt schwarz auf weiß erscheinen. Sie können dann nicht ausgewählt werden. Unter einem grauen Menü-Titel in der Menü-Leiste wird bei einem Maus-Klick kein Menü hervorklappen, und ein grauer Menü-Punkt wird nicht invertiert werden, wenn man mit der Maus darüberfährt. Weiterhin können einzelne Menü-Punkte auch ohne die Maus ausgewählt werden, indem eine bestimmte Taste (die Befehls- oder *Command*-Taste) festgehalten wird und gleichzeitig eine andere Taste gedrückt wird, die bestimmt, welcher Punkt aus welchem Menü ausgewählt wird. Dies nennt man dann ein "Tastatur-Äquivalent" dieses Menü-Punktes. Gerade Tastatur-Äquivalente entscheiden oft zwischen einem komfortablen, aber umständlichen Programm, das man bald links liegenläßt

und einem komfortablen und schnellen Programm, das man auch dann noch gerne benutzt, wenn man es schon länger kennt.

Die ganze Handhabung eines Menüs sieht beim fertigen Programm recht einfach und mühelos aus, ist aber mit beträchtlichem Programm-Aufwand verbunden. Dies ist leicht zu verstehen, wenn man ein wenig über die Vorgänge nachdenkt, die programmtechnisch wohl nötig sind, ein Menü auf dem Bildschirm zu zeichnen und die Benutzeraktionen dann korrekt zu verarbeiten. Es gibt ja außer den oben beschriebenen Regeln für eine Menü-Auswahl noch eine ganze Reihe weiterer Prinzipien, da alle wohlüberlegt sind und aus langjährigen Versuchen von Apple und XEROX hervorgegangen sind. Dank der ToolBox ist dieser ganze komplexe Vorgang aber so einfach, wie er für den Anwender des fertigen Programms auch aussieht.

5.2 Menüs aus der Sicht des Programmierer

5.2.1 Nummern und Handles

Für die meisten Teile eines Programms sind Menüs und Menü-Punkte nur Zahlen. Jedes Menü besitzt eine eindeutige Zahl (die *Menu ID* oder Menü-Nummer), die ihm beim Erzeugen zugeordnet wird, und die Menü-Punkte werden in jedem Menü einfach von oben durchnumeriert. Hat eine ToolBox-Prozedur festgestellt, daß eine Menü-Auswahl getroffen wurde, so teilt sie dies dem Programm durch diese beiden Zahlen mit: die Menü-Nummer und die (laufende) Nummer des gewählten Punktes.

Die Datenstruktur, hinter der sich eine Menü-Leiste bzw. ein einzelnes Menü verbirgt, ist natürlich wesentlich komplizierter, aber für die wenigsten Programme von Interesse. Normalerweise hat man nur mit Handles auf diese Datenstrukturen zu tun.

```
TYPE
     MenuPtr    = ^MenuInfo;
          {MenuInfo wird hier nicht näher erläutert}
     MenuHandle= ^MenuPtr;
```

MenuHandles werden nur dann benötigt, wenn man Menüs erzeugt, löscht oder verändert. Hinter ihnen verbirgt sich das gesamte Aussehen eines einzelnen PullDown-Menüs, so z.B. die Texte aller Menü-Punkte und des Menü-Titels. Nachdem ein neues Menü erzeugt wurde, wird dessen Handle

normalerweise an den MenuManager übergeben. Dieser trägt es in die interne Menü-Liste ein, in der alle gerade sichtbaren Menüs verzeichnet sind, und sorgt dann dafür, daß der Menü-Titel in der Menü-Leiste erscheint und die Menü-Punkte vom Benutzer ausgewählt werden können.

Wenn ein Programm ein Menü erst einmal in Form des entsprechenden Handles an den MenuManager übergeben hat, benötigt es diesen meist nicht mehr. Er braucht deshalb auch nicht in einer Variablen gespeichert zu werden. Hat der Anwender eine Auswahl aus einem Menü getroffen, so meldet dies der MenuManager unter Angabe der Menü-Nummer und der Nummer des gewählten Punktes; der **MenuHandle** wird dazu nicht benötigt.

Menü-Nummer
1 256 257 261 265
Ablage Bearbeiten Auffinden Format
1 Nicht widerrufbar ⌘Z
2
3 Ausschneiden ⌘X
4 Kopieren ⌘C
Punkt-Nummer 5 Einsetzen ⌘V
6
7 Zwischenablage

Bild 5 - 2: *Menü-Nummern und Menü-Punkte*

5.2.2 Funktionen zum Erzeugen von Menüs

Normalerweise werden Menüs beim Start eines Programms erzeugt und werden danach nur noch abgefragt. Guter Stil ist es zudem, Menüs in Resources zu legen und keinen Text — z.B. für einen Menü-Titel oder einen Menü-Punkt — im Programm erscheinen zu lassen. Prinzipiell macht es jedoch für den Benutzer keinen Unterschied, ob das Aussehen eines Menüs im Resource-Teil eines Programms festgelegt wird oder im Programmtext.

```
FUNCTION NewMenu
                  (menuID: INTEGER;
                   menuTitle: Str255
                  ): MenuHandle;
```

Hierdurch wird ein neues Menü erzeugt und ein Handle darauf zurückgegeben. Dieses Menü ist zunächst leer und besitzt keine Menü-Punkte. Es erscheint auch nicht am Bildschirm, solange es nicht mit **InsertMenu** (s.u.) in die Menü-Liste eingesetzt wurde.

```
FUNCTION GetMenu(resourceID: INTEGER): MenuHandle;
```

Entspricht in seiner Funktion im wesentlichen einem Aufruf von **NewMenu**. Nur werden alle Informationen für den Aufbau des Menüs aus dem Resource-Teil des Programms geholt und brauchen nicht im Programm angegeben werden. Die Angaben in der Resource bestimmen dabei sowohl den Titel, die Nummer und die einzelnen Punkte des Menüs.

```
PROCEDURE AppendMenu
                  (theMenu: MenuHandle;
                   data: Str255);
```

AppendMenu fügt (weitere) Menü-Punkte unten an die schon vorhandenen Punkte des Menüs, auf das der Handle **theMenu** zeigt, an. Wie diese Punkte aussehen, wird im Text **data** angegeben. Das Format für **data** sieht dabei aus wie folgt:

Data kann die Angaben für ein oder mehrere Menü-Punkte enthalten. Um diese unterscheiden zu können, werden die Angaben für jeden Menü-Punkt durch ein Semikolon ';' von den Angaben für den folgenden getrennt. Für jeden Menü-Punkt muß der Text angegeben werden, der für diesen Punkt im Menü erscheinen soll, danach folgen optional die weiteren Informationen, die das Aussehen dieses Punktes bestimmen.

Diese zusätzlichen Informationen beginnen mit sog. "Meta-Buchstaben". Sie heißen *Meta*-Buchstaben, da sie nicht im Text erscheinen, sondern nur das Aussehen des Punktes ändern. Meta-Buchstaben sind immer Sonderzeichen, um sie sofort als besondere Information kenntlich zu machen.

Die Schrift, in der der Text dargestellt wird, ist immer Chikago-12. Jeder Punkt kann aber in einem anderen Schriftstil gezeigt werden. Erscheint in den Angaben zu einem Menüpunkt das Kleiner-Zeichen '<' so bestimmt der darauf folgende Buchstabe den Zeichen-Stil dieses Punktes:

<B	=	**fetter Text**
<I	=	*kursiver Text*
<U	=	unterstrichener Text
<O	=	Umrahmter Text
<S	=	Schattierter Text

Es kann jedoch für jeden Menü-Punkt nur eine dieser Angaben gemacht werden. Mit **AppendMenu** ist es nicht möglich, z.B. fette und unterstrichene Menü-Punkte zu spezifizieren.

Möchte man, daß ein Menü-Punkt auch über ein Tastatur-Äquivalent ausgesucht werden kann, so gibt man einen Schrägstrich '/' und dahinter den entsprechenden Buchstaben, durch den der Punkt gewählt wird, an. Enthält **data** bei einem **AppendMenu**-Aufruf z.B. den Text "Löschen/B", so kann der Menü-Punkt mit dem Namen "Löschen" auch aufgerufen werden, indem man die Befehls-Taste festhält und die Taste "B" drückt. Hierfür sollten übrigens am besten nur Großbuchstaben und keine Umlaute oder Sonderzeichen verwendet werden, da dies zu meist unerwünschten Schwierigkeiten mit ausländischen Tastaturen führt.

Möchte man, daß ein Menü-Punkt zunächst nicht ausgewählt werden kann, so fügt man den entsprechenden Angaben der Buchstaben eine offene Klammer '(' hinzu. Dieser Menü-Punkt erscheint dann im Menü grau und kann weder mit der Maus noch mit einem eventuell vorhandenen Tastatur-Äquivalent ausgewählt werden. Einen grauen Strich, durch den man verschiedene Teile eines Menüs trennen kann, erhält man z.B. durch den Text '(-'.



Bild 5 - 3: *Erzeugung eines Menüs*

Das folgende Programm-Stück ergibt z.B. ein Menü, das hinterher auf dem Bldschirm aussieht wie Bild 5 - 3.

```
meinMenu := NewMenu(1,'Bearbeiten');
AppendMenu(meinMenu,
               '(Nicht widerrufbar/Z;
               (-;
               Ausschneiden/X;
               Kopieren/C;
               Einsetzen/V;
               (-;
               Zwischenablage');
```

Menü-Punkte können noch weitere Eigenschaften haben und können auch ganz von der Text-Form abweichen. Diese Eigenschaften werden aber nur selten benötigt oder verlangen weitergehendes Wissen über die ToolBox und sollen deshalb hier nicht erörtert werden, sie können bei Bedarf im Kapitel *MenuManager* von *Inside Macintosh* nachgelesen werden.

```
PROCEDURE InsertMenu
               (theMenu: MenuHandle;
                before: INTEGER);
```

Nachdem durch Aufrufe von **NewMenu** und **AppendMenu** (bzw. **GetMenu**) ein komplettes Menü geformt wurde, trägt **InsertMenu** es in die Menü-Liste des MenuManagers ein. Es wird dabei auf dem Bildschirm links

neben dem Menü erscheinen, das die Nummer **before** trägt. Übergibt man 0 für **before,** so wird es rechts an das Ende der Menü-Liste angehängt. Es wird aber zunächst nur in die Menü-Liste eingetragen und erscheint noch nicht auf dem Bildschirm!

```
PROCEDURE DrawMenuBar;
```

zeichnet die Menü-Leiste entsprechend dem aktuellen Zustand der Menü-Liste auf den Bildschirm. Nachdem irgendwelche Operationen ausgeführt wurden, die den Inhalt der Menü-Liste geändert haben, sollte stets **DrawMenuBar** aufgerufen werden. Nachdem man z.B. die einzelnen Menüs eines Programms erzeugt hat und die entsprechenden Handles mit **InsertMenu** in diese Liste eingetragen hat, sollte am Ende unbedingt einmal **DrawMenuBar** aufgerufen werden.

Zusammenfassend kann man sagen, daß in der Initialisierungsphase eines Programms zunächst **InitMenus** aufgerufen wird. Danach werden alle Menüs durch Aufrufe von **NewMenu** und **AppendMenu** bzw. **GetMenu** gebildet. Die entsprechenden **MenuHandles** werden dann nacheinander mit **InsertMenu** in die Menü-Liste eingetragen und die daraus resultierende Menü-Leiste wird durch einen Aufruf von **DrawMenuBar** gezeichnet.

5.2.3 Das Modifizieren von Menüs

Wie schon mehrfach erwähnt, wird üblicherweise das Aussehen von Menüs beim Start des Programms bestimmt und danach nicht mehr geändert. In seltenen Fällen ist es jedoch nötig, ein bereits auf dem Bildschirm sichtbares Menü nachträglich zu modifizieren. Ein relativ häufiger Fall, in dem dies nötig wird, ist das Aktivieren und Deaktivieren (Grauzeichnen) einzelner Menü-Punkte. Manchmal können verschiedene Menü-Befehle nur aufgerufen werden, wenn vorher bestimmte Voraussetzungen geschaffen wurden. Gelten diese Voraussetzungen nicht, sollte der entsprechende Menü-Punkt deaktiviert werden. Eine anderer Fall, in dem nachträgliche Änderungen nötig werden, ist das Versehen eines Menü-Punktes mit einem Haken, um anzuzeigen, daß ein bestimmter Programmzustand besteht, z.B. ein bestimmter Zeichensatz gewählt wurde. Prinzipiell kann man allerdings mit den folgenden Funktionen alle Aspekte des Aussehens von Menüs und Menü-Punkten nachträglich ändern — man sollte es jedoch nicht allzu häufig tun, um den Benutzer nicht zu verwirren!

Zur Modifikation eines Menüs bzw. Menü-Punktes wird der entsprechende **MenuHandle** benötigt — die Menü-Nummer reicht nicht! Es ist jedoch

möglich, den **MenuHandle** eines Menüs, das im Resource-Teil eines Programms definiert ist, jederzeit wiederzufinden, indem man die Funktion **GetMenu** mit der entsprechenden Nummer aufruft. Will man dies vermeiden, ist es ratsam, die Handles für alle Menüs, die ein Programm verwendet, in Variablen zu speichern (z.B. in einem ARRAY).

```
PROCEDURE SetItem
                (theMenu:     MenuHandle;
                 item:        INTEGER;
                 itemString: Str255);
```

ändert den Text des Menü-Punktes **item** im Menü **theMenu** auf **itemString**. **SetItem** ändert immer nur den Text eines Menü-Punktes. Selbst wenn **itemString** Meta-Buchstaben enthält, hat dies keine Auswirkung auf das Aussehen des Menü-Punktes — sie erscheinen im Text des Menü-Punktes. Meta-Buchstaben wirken nur bei **AppendMenu**!

```
PROCEDURE GetItem
                (theMenu:        MenuHandle;
                 item:           INTEGER;
                 VAR itemString: Str255);
```

erlaubt es, nachträglich den Text eines Menü-Punktes vom Programm aus festzustellen. **ItemString** enthält nach Aufruf von **GetItem** den Text des entsprechenden Menü-Punktes — aber keine weiteren Informationen über das Aussehen dieses Punktes (wie z.B. den Text-Stil).

```
PROCEDURE DisableItem
                (theMenu: MenuHandle;
                 item: INTEGER);

PROCEDURE EnableItem
                (theMenu: MenuHandle;
                 item: INTEGER);
```

DisableItem deaktiviert den Punkt **item** im Menü **theMenu**; **EnableItem** aktiviert ihn. Deaktivierte Menü-Punkte erscheinen grau statt schwarz und können nicht mit der Maus oder über die Tastatur ausgewählt werden. Übergibt man für **item** 0, so wird das ganze Menü aktiviert bzw. deaktiviert. Die Titel von deaktivierten Menüs und alle Punkte dieses Menüs erscheinen grau und können nicht gewählt werden. Der graue Menü-Titel erscheint aber erst dann statt des schwarzen auf dem Bildschirm, wenn **DrawMenuBar** aufgerufen wird!

```
PROCEDURE CheckItem
                (theMenu: MenuHandle;
                 item: INTEGER;
                 checked: BOOLEAN);
```

CheckItem versieht einen Menü-Punkt mit einem Haken oder entfernt diesen Haken wieder. Ist **checked** TRUE, so erscheint der Haken links neben dem Menü-Text, andernfalls bleibt der Platz leer. Eine typische Anwendung für solche Haken sind Menüs, in denen der Anwender eine Auswahl unter mehreren verschiedenen Einstellungen für einen Programmparameter treffen kann. Die gerade gültige Einstellung sollte immer mit einem Haken versehen sein, abgehakt werden.

Das waren die wichtigsten Arten, ein Menü nachträglich zu modifizieren. Mit anderen Prozeduren ist es jederzeit nachträglich möglich, den Text-Stil von Menüs beliebig zu modifizieren, kleine Bilder (Icons) in Menüs erscheinen zu lassen und Menü-Punkte noch mit anderen Buchstaben, als nur dem Haken zu markieren.

5.2.4 Das Abfragen von Menüs

Wenn erst einmal alle Menüs definiert sind und in die Menü-Liste eingesetzt sind, kann der Benutzer seine Auswahl treffen. Eine Menü-Auswahl wird stets durch einen Klick in die Menü-Leiste oder durch einen Tastendruck bei festgehaltener Befehls-Taste eingeleitet. Sobald ein Anwendungsprogramm eines dieser beiden Ereignisse registriert, gibt es die Kontrolle an den MenuManager ab. Dies geschieht, indem eine von zwei Funktionen des MenuManagers aufgerufen werden (je nachdem ob ein Maus-Klick oder ein Tastendruck vorliegt). Aus dem Funktionsergebnis läßt sich ablesen, ob der Benutzer wirklich einen Menü-Punkt ausgewählt hat und welchen.

```
FUNCTION MenuSelect (startPt: Point): LONGINT;
```

Sobald ein Programm einen Maus-Klick in der Menü-Leiste entdeckt hat, sollte es **MenuSelect** aufrufen und als Parameter die Position des Maus-Klicks in globalen Koordinaten übergeben. **MenuSelect** behält so lange die Kontrolle, bis die Maus-Taste gelöst wird. Das Herunterklappen von Menüs und Invertieren von Menü-Punkten wird alles von **MenuSelect** erledigt; das Anwendungsprogramm braucht sich darum nicht zu kümmern. Befindet sich der Cursor zu dem Zeitpunkt des Lösens der Maus-Taste in einem aktivierten Menü-Punkt, so werden die Menü-Nummer und die Nummer des Punktes als Funktionswert zurückgeliefert. Diese beiden 16 Bit großen Zahlen werden

dabei in den 32-Bit-Funktionswert "gepackt". Die höherwertigen 16 Bits enthalten die Menü-Nummer und die niederwertigen 16 Bit die Nummer des Menü-Punktes, der gewählt wurde. Mit den beiden Hilfs-Funktionen **HiWord** und **LoWord** kann das Funktions-Ergebnis in die beiden benötigten 16-Bit-Zahlen zerlegt werden. Das entsprechende Programmstück sieht üblicherweise aus wie folgt:

```
Bit32    := MenuSelect(mousePt);
theMenu  := HiWord(Bit32);
theItem  := LoWord(Bit32);
```

Wenn der Benutzer keine Auswahl aus einem Menü getroffen hat, nachdem er mit der Maus in die Menü-Leiste geklickt hat, liefert **MenuSelect** als Ergebnis 0 zurück.

```
FUNCTION MenuKey (ch: CHAR): LONGINT;
```

Drückt der Anwender auf eine Taste und hält gleichzeitig die Befehls-Taste fest, sollte ein Programm **MenuKey** aufrufen, in der Annahme, daß dies ein Tastatur-Äquivalent für die Auswahl eines Menü-Punktes ist. Existiert wirklich ein Menü-Punkt in irgendeinem Menü, der das Tastatur-Äquivalent **ch** hat, liefert **MenuKey** dasselbe Ergebnis zurück, das **MenuSelect** geliefert hätte, wenn derselbe Menü-Punkt mit der Maus ausgewählt worden wäre.Wenn kein Menü-Punkt existiert, der über das Tastatur-Äquivalent **ch** ausgewählt werden kann, liefert **MenuKey** als Ergebnis 0 zurück.

Wenn **MenuSelect** oder **MenuKey** ein Ergebnis ungleich 0 hatten, so sorgt der MenuManager dafür, daß nach dem Aufruf einer der beiden Prozeduren der entsprechende Menü-Titel invertiert ist. Dies soll dem Benutzer anzeigen, daß der Befehl erfolgreich ausgewählt wurde und nun ausgeführt wird. Solange die Ausführung des Befehls nicht beendet ist, sollte der Titel auch invertiert bleiben, danach sollte es wieder normal (schwarz auf weiß) aussehen. Da der MenuManager aber nicht wissen kann, wann die Befehls-Ausführung beendet ist, muß das Anwendungsprogramm dafür sorgen, daß der Menü-Titel wieder sein normales Aussehen erhält. Dies ist die Hauptanwendung der Funktion **HiliteMenu**.

```
PROCEDURE HiliteMenu (menuID: INTEGER);
```

HiliteMenu sorgt dafür, daß der Titel des Menüs mit der Nummer **menuID** invertiert wird. Da immer nur ein Menü-Titel invertiert sein darf, werden zuvor alle anderen normal gezeichnet. Übergibt man für **menuID** 0 oder eine Nummer, die kein Menü in der Menü-Liste besitzt, so bewirkt dies nur ein

normales Aussehen aller Menü-Titel. Diese Prozedur sollte üblicherweise nach der Beendigung der Ausführung eines Befehls aufgerufen werden, der durch eine Menü-Auswahl erfolgte.

5.3 Schlußbemerkung

Wie wir gesehen haben, nimmt der MenuManager dem Programm den Großteil der Arbeit im Zusammenhang mit den PullDown-Menüs des Macintosh ab. Die Prozeduren **MenuSelect** und **MenuKey** realisieren das ganze Standardverhalten solcher Menüs und brauchen nur vom Anwendungsprogramm im richtigen Moment aufgerufen werden. Welcher Moment das ist, werden wir im Kapitel *Ereignisse* bzw. im Teil II dieses Buches noch genauer erfahren.

6 Ereignisse

Es mag vielleicht ein wenig komisch erscheinen, im Zusammenhang mit einem Computer-Programm von Ereignissen zu sprechen; genau sie sind es aber, die die Essenz eines Macintosh-Programms ausmachen. Ein Macintosh-Programm tut — wie wir alle — den größten Teil seiner Zeit eigentlich nichts anderes, als darauf zu warten, daß irgend etwas passiert. Auf dem Macintosh wartet es zwar mit der phantastischen Geschwindigkeit des 16-Bit-Prozessors M68000, tut aber sonst nichts. Erst wenn "etwas" passiert — eben ein Ereignis eintritt — beginnt die hektische Arbeit, werden die "eigentlichen" Funktionen des Programms ausgeführt. Das entsprechende Programmstück, das dies realisiert, sieht ungefähr so aus:

```
repeat
     ist_Etwas_Passiert;
     Wenn_Ja:
          prüfe_das_Ereigniss;
          handle_entsprechend;
until forever;
```

Der Macintosh kann natürlich nicht alle Ereignisse bemerken, die in seiner Umgebung passieren, sondern nur die, die er mit den Sinnesorganen, die man ihm eingebaut hat, registrieren kann. Dies sind im wesentlichen das Drücken einer Taste auf der Tastatur und das Drücken der Taste auf der Maus. Ansonsten ist ein Macintosh blind. Es gibt noch eine ganze Reihe anderer Ereignisse, auf die ein Macintosh-Programm reagieren muß, diese haben jedoch meist nichts mit der Außenwelt zu tun, sondern entsprechen wichtigen Ereignissen in seinem Innern und auf seinem Bildschirm.

Darin, daß es die meiste Zeit nichts tut und auf irgendein Ereignis wartet, unterscheidet sich ein Macintosh-Programm noch nicht von anderen Programmen auf gewöhnlichen Computern. Auch bei diesen ist es üblich, z.B. auf den nächsten Tastendruck zu warten, wenn das Programm eine Eingabe erwartet. Man könnte meinen, daß das einzige, was diese Situation auf dem Macintosh etwas verkompliziert, die Maus ist, da man deren Taste noch zusätzlich drücken kann. Aber der wesentliche Unterschied ist ein ganz

anderer. Programme, die z.B. eine Text-Eingabe erwarten, befinden sich auf anderen Computern üblicherweise in einem Modus. Das Warten auf Tastendrücke und ähnliches wird bei diesen Programmen an vielen Stellen durchgeführt, und die Folgen sind abhängig von der (Programm-)Stelle, wo gewartet wird. Ein Tastendruck (oder Maus-Klick) in einem Moment wird meist eine ganz andere Wirkung haben, wie wenige Augenblicke später. Ein Modus aber ist nach der Philosophie des Macintosh etwas abgrundtief Böses.

6.1 Don't Mode Me In!

Die Grundidee, die hinter dem Macintosh steht, ist es, die Bedienung von Computern so leicht wie möglich zu machen. Und der feste Glaube der Designer des Macintosh war es, daß ein Programm-Modus die Bedienung erschwert. Besitzt ein Programm verschiedene Modi, so muß sich der Anwender nicht nur die zur Bedienung des Programms nötigen Befehle merken, sondern auch noch darauf achten, wo er sich im Programm gerade befindet, um die Wirkung seiner Befehle voraussehen zu können. Der Schlachtruf der Macintosh-Entwickler (und ihrer vielen Vorkämpfer) lautete deshalb "Don't Mode Me In!" (zu deutsch ungefähr "Zwing mir keinen Modus auf").

Der Macintosh kommt weitgehend ohne Modi aus. Es ist fast zu jedem Zeitpunkt möglich, das aktive Fenster zu wechseln oder einen Befehl aus der Menü-Leiste herabzuholen. Diese Befehle bewirken zu einem großen Teil in jedem Moment des Programmablaufs dieselbe Reaktion des Systems. Ein anderer Teil der Befehle wirkt verschieden, je nachdem, welches Fenster gerade oben liegt und was man in diesem Fenster mit der Maus selektiert hat. Es gilt aber immer die Regel, daß vergleichbare Aktionen des Benutzers vergleichbare Reaktionen des Computers hervorrufen.

Um dieses Verhalten eines Programms zu erreichen, ist es sinnvoll, die Behandlung aller Ereignisse an zentraler Stelle im Programm zu handhaben. Hierdurch wird vermieden, daß Programmteile doppelt und dreifach geschrieben werden müssen, wenn man an verschiedenen Stellen Eingaben erwartet. Noch viel wichtiger ist aber, daß dadurch auch nicht gleiche Ereignisse an unterschiedlichen Programm-Stellen unterschiedlich behandelt werden (das wäre ein Modus). Hierdurch kommt die etwas merkwürdige Programmstruktur zustande, die Macintosh-Programme üblicherweise haben.

Das Hauptprogramm (in der Pascal-Terminologie) enthält im wesentlichen nur das Warten auf Ereignisse, das Klassifizieren der Ereignisse und dann

den Aufruf der entsprechenden Prozeduren, die auf diese Ereignisse reagieren. Große Teile dieses Hauptprogramms sind dabei für sehr unterschiedliche Programme völlig identisch. Sie realisieren das Standard-Verhalten, das jedes Macintosh-Programm zeigen muß, und brauchen deshalb auch nur einmal im Leben geschrieben werden und können immer wiederverwendet werden. Im Gegensatz zur üblicherweise für große Programme stark propagierten hierarchischen Programmstruktur, kümmert sich hier das Hauptprogramm selbst um stark Hardware-abhängige Details, wie z.B. einen einzelnen Tastendruck. Dies (und die Fensterverwaltung) erfordert ein starkes Umdenken — gerade auch für schon erfahrene Programmierer.

Hat man diese Prinzipien erst einmal begriffen, ist die Programmierung allerdings auch nicht schwerer als auf anderen Rechnern. Wichtig ist dabei vor allem der Programm-Rahmen, in dem die Behandlung der Ereignisse geschieht. Ist er erst einmal fertig und arbeitet korrekt, so kann man in diesen Rahmen die anderen Teile des Programms relativ problemlos "einhängen". Der Rahmen an sich ist weitgehend wiederverwendbar und bedeutet beim nächsten Programm deshalb keine neue Arbeit mehr. Fast von selbst folgt aus der Verwendung eines korrekten Rahmens dann ein ereignisorientiertes Programm ohne Modi.

6.2. Die verschiedenen Ereignisse

Der für die Verwaltung von Ereignissen zuständige Teil in der Macintosh-ToolBox ist der **EventManager**. Ereignisse werden auf englisch "*events*" genannt, und ich werde den deutschen Ausdruck "Ereignis" und den englischen "event" parallel verwenden. Der EventManager kennt 16 verschiedene Typen von Ereignissen und kann auch nur diese 16 verwalten. Die Ereignisse sind dabei einfach von 0 bis 15 durchnumeriert. Da Zahlen aber wenig aussagekräftig sind, ist in *Inside Macintosh* für jeden Ereignis-Typ auch noch eine Konstante definiert, die dem entsprechenden Ereignis-Typ eine mnemonische Bezeichnung gibt. Von diesen 16 Typen werden allerdings im Moment nur 12 von der ToolBox selbst verwendet, die restlichen sind für eine Verwendung durch Anwendungsprogramme reserviert (was allerdings recht unüblich ist). Und von den 12 von der ToolBox verwendeten Ereignis-Typen wiederum sind nur 5 bzw. 6 wirklich wichtig für die meisten Programme. Die anderen sind nur für sehr spezielle Anwendungen von Bedeutung.

Die folgenden Konstanten für Event-Typen sind im EventManager definiert:

```
CONST
     nullEvent =      0;
     mouseDown =      1;
     mouseUp =        2;
     keyDown =        3;
     keyUp =          4;
     autoKey =        5;
     updateEvt =      6;
     diskEvt =        7;
     activateEvt =    8;
     { Event-Typ 9 ist für
          künftige Erweiterungen reserviert }
     networkEvt =     10;
     driverEvt =      11;
     { Event-Typen 12 - 15
          reserviert für das Anwendungsprogramm }
```

Ein **nullEvent** wird immer dann an das Programm gemeldet, wenn keine anderen Ereignisse vorliegen. Dies ist ein Signal an das Programm, daß nichts zu tun ist, und kann irgendwelche Aktionen einleiten, die immer dann ausgeführt werden, wenn keine Aufträge vom Benutzer vorliegen — z.B. das Blinken des Einfügestrichs im Text oder die Änderung der Cursor-Form.

Einer der wichtigsten und kompliziertesten Events ist der **mouseDown**-Event. Ein solches Ereignis entsteht – wie schon der Name sagt – dadurch, daß der Benutzer die Maustaste herunterdrückt. Durch Maus-Klicks können eine Vielzahl verschiedener Wünsche des Benutzers ausgedrückt werden. Klickt er in der Menü-Leiste, wird er meist einen Menü-Befehl auswählen wollen, klickt er auf ein Fenster, so leitet dies eine der vielen Aktionen ein, die im Kapitel über den WindowManager beschrieben wurden usw. Dem **MouseDown**-Event wird weiter unten noch ein ganzer Abschnitt gewidmet.

Ein **mouseUp**-Event ist für die wenigsten Programme von Interesse und wird hier nur der Vollständigkeit halber mit aufgeführt. Er entsteht durch das Lösen der Maus-Taste (wird im allgemeinen also immer hinter einem **mausDown**-Event vorliegen).

Ein **keyDown**-Event entsteht durch das Drücken einer Taste auf der Tastatur. Ausnahmen machen hier nur die Shift-, Befehls- und Auswahl-Taste, die alleine keine Ereignisse bedeuten. Erst wenn sie zusammen mit einer anderen Taste gedrückt werden, bedeutet dies ein Ereignis. Eventuell

können auch zwei Tastendrücke hintereinander einer **keyDown**-Event bedeuten. Drückt man eine der Accent-Tasten und danach eine andere Taste, so bewirkt dies bei Vokalen meist nur ein Ereigniss statt zweien. Dieses Ereignis entspricht dann dem Drücken einer entsprechend akzentuierten Taste, die es auf der deutschen Tastatur nicht gibt. So können z.B. auch Umlaute auf der englischen Tastatur eingegeben werden.

KeyUp-Events bedeuten ähnlich wie **mouseUp**-Events für die meisten Programme nichts und entstehen beim Lösen einer zuvor gedrückten Taste.

AutoKey-Events entsprechen einem "normalen" Druck auf eine Taste, nur daß sie nicht durch einen wirklichen Tastendruck hervorgerufen wurden, sondern vom System erzeugt werden, wenn eine Taste längere Zeit festgehalten wird (*Auto-Repeat*-Funktion). Sie haben deshalb einen eigenen Typ, da sie manchmal anders behandelt werden müssen als "normale" Tastenbetätigungen.

Ebenso wie **autoKey**-Events werden auch **updateEvts** vom Betriebssystem erzeugt und kommen nicht aus der Außenwelt. Immer wenn ein Teil eines Fensters neu gezeichnet werden muß, generiert das System einen **updateEvt**, um dies dem Programm mitzuteilen.

Ein **diskEvt** teilt dem Programm mit, daß eine Diskette neu in eines der Laufwerke eingelegt wurde. Dies ist nur für die wenigsten Programme von Interesse.

Ein **activateEvt** ist für die meisten Programme schon wichtiger, da er dem Programm mitteilt, daß ein Fenster aktiv geworden ist, das vorher hinten lag oder ein aktives Fenster vom aktiven zum inaktiven Zustand übergeht. Dies ist vor allem dann wichtig, wenn eine Reihe von Menübefehlen nur dann sinnvoll sind, wenn ein bestimmtes Fenster aktiv ist. Zudem ändert sich das Aussehen mancher Fenster bzw. ihres Inhalts, je nachdem, ob sie vorne liegen (aktiv sind) oder hinter anderen Fenstern (inaktiv sind).

Die beiden Typen **networkEvt** und **driverEvt** sind nur für wenige Programme interessant und sollen deshalb nicht näher behandelt werden

6.3 Der EventRecord

Wenn ein Programm auf ein Ereignis reagiert, so genügt es selbstverständlich nicht, nur den Typ dieses Ereignisses zu kennen. Bei einem **mouseDown**-

Event ist es z.B. wichtig, auch zu erfahren, bei welcher Position sich die Maus befand, als die Taste gedrückt wurde, und für **keyDown**-Events möchte man schließlich auch genau wissen, welche Taste denn nun gedrückt wurde usw. Deshalb kommuniziert ein Anwendungsprogramm auch mit einer wesentlich komplizierteren Datenstruktur mit dem EventManager als nur einer Zahl. Ein Ereignis wird dem Programm in Form eines **EventRecords** mitgeteilt, der folgenden Aufbau hat:

```
TYPE EventRecord =
    RECORD
      what:      INTEGER; { Der Typ von 0 - 15      }
      message:   LONGINT;
                  { Inhalt; abhängig vom Typ        }
      when:      LONGINT;
                  { Zeit, als das Ereignis eintrat}
      where:     Point;
                  { Maus-Pos. zum Zeitpunkt when    }
      modifiers: INTEGER;{ Statusinformation       }
    END {EventRecord};
```

Die Felder eines EventRecord enthalten alle Informationen, die das Programm zur Bearbeitung eines Ereignisses vom entsprechenden Typ benötigt (zumindest alle Informationen, die einem das Betriebssystem dazu liefern kann).

6.3.1 Überblick über die Felder des EventRecords

What enthält den Typ des Ereignisses, den dieser EventRecord beschreibt.

Message enthält zusätzliche Informationen, die einem das Betriebssystem zur Bearbeitung eines Events mitteilt. Sein Inhalt wird von vielen Events überhaupt nicht benötigt und ist stark abhängig vom Typ des Ereignisses. Auf dieses Feld gehe ich weiter unten noch näher ein.

When enthält den genauen Zeitpunkt, zu dem das Ereignis eintrat. Dieser Zeitpunkt wird gemessen in 60stel Sekunden seit dem letzten Starten des Macintosh und hat also wenig mit der Uhrzeit zu tun. Er kann aber gut für eine Messung des zeitlichen Abstands zweier Ereignisse verwendet werden (z.B. zum Prüfen auf einen Doppel-Klick). Sonst hat **when** kaum eine Bedeutung.

Where enthält die Position des Maus-Cursors in globalen Koordinaten zum Zeitpunkt, als das Ereignis passierte (**when**). Liegen diese Koordinaten im Innern eines Fensters, so muß man sie erst mittels **GlobalToLocal** ins lokale Koordinaten-System des Fensters umsetzen, bevor man weiter damit arbeitet.

Modifiers enthält eine Reihe von Informationen vor allem über den Zustand der Tastatur zum Zeitpunkt, als das Ereignis eintrat. Diese Informationen sind bitweise in der Zahl **modifiers** "verpackt" und werden im übernächsten Abschnitt genauer erläutert.

6.3.2 Die Bedeutung der Event.message

Viele Events werden ausreichend durch ihren Typ (im Feld **what**) beschrieben. Andere benötigen weitere Informationen zu ihrer Behandlung. Diese Informationen werden dem Programm im Feld **message** mitgeteilt.

Bei **keyDown**-, **keyUp**- und **AutoKey**-Events enthält **message** Informationen über die gedrückte bzw. gelöste Taste. Hierzu werden nur die niederwertigen 16 Bit der 32 Bit von **message** benötigt. Diese 16 Bit zerfallen in zwei Felder, von denen für die meisten Programme nur eines interessant ist. Die Bits 0–7 enthalten den *character code*, den (erweiterten) ASCII-Wert des Buchstabens, der zu der gedrückten Taste gehört. Drückt man auf die Taste 'g', so enthalten diese 8 Bit die Zahl 103, hält man gleichzeitig die Shift-Taste fest, ist der Wert gleich 71 usw.

Dies reicht für die meisten Anwendungen vollkommen aus. Benötigt man jedoch genauere Informationen darüber, welche Taste gedrückt wurde, unabhängig von dem Buchstaben, mit dem diese Taste belegt ist, enthält man diese Information in den Bits 8-15 der **message**. Die hier "versteckte" Zahl ist der sog. *key code*, eine eindeutige Zahl, die jeder Taste auf der Tastatur zugeordnet ist. Deren Werte können in *Inside Macintosh* nachgeschlagen werden, wenn sie von Interesse sind. Es ist nicht besonders ratsam, mit *key codes* zu arbeiten, weil sich die Zuordnungen von *key codes* zu *character codes* und auch die Tasten-Positionen auf der Tastatur von Land zu Land ändern können.

Bei **activateEvts** und **updateEvts** enthält **message** einen Zeiger (WindowPtr) auf das Fenster, das von diesem Event betroffen ist. Mit Hilfe dieses Zeigers kann dann bestimmt werden, welche Aktionen unternommen werden müssen, um auf den Event zu reagieren.

6.3.4 Der Aufbau von Event.modifiers

Modifiers enthält eine Reihe von Informationen, die unabhängig vom Event-Typ registriert werden und eine eventspezifische Information. Bei einem **activateEvt** enthält das unterste Bit die Information, ob das betroffene Fenster vom aktiven in den inaktiven Zustand übergeht oder umgekehrt. Ist dieses Bit an (=1), so wird das Fenster, auf das **message** zeigt, aktiv. Ist das Bit = 0, so wird das Fenster inaktiv. Die restlichen Bits werden wie folgt belegt (die Numerierung für die Bits beginnt dabei mit 0 für das niederwertigste Bit):

Bit 7= 1, wenn die Maus-Taste zum Zeitpunkt des Ereignisses gedrückt war.
Bit 8= 1, wenn die Befehls- (*Command-*)-Taste gedrückt war.
Bit 9= 1, wenn die Shift-Taste gedrückt war.
Bit10=1, wenn die Shift-Lock-Taste eingerastet war.
Bit11=1, wenn die Wahl-(*Option-*)Taste gedrückt war.

Alle anderen Bit-Positionen (1–6 und 13–15) sind im Moment nicht belegt und für künftige Erweiterungen reserviert. Für die meisten Programme ist außerdem meist nur das Bit 8 und manchmal das Bit 9 interessant. Bei Tastatur-Events kann die gedrückte Befehls-Taste (Bit 8) die Auswahl eines Menü-Punktes durch ein Tasten-Äquivalent bedeuten. Und ein Drücken der Maus-Taste mit gedrückter Shift-Taste (auch "Shift-Klick" genannt) hat manchmal eine etwas andere Bedeutung als ein Maus-Klick ohne gedrückte Shift-Taste.

6.4 Wie bemerkt man ein Ereignis ?

Wir alle haben es nicht gerne, wenn etwas Wichtiges passiert und wir merken nichts davon. Genauso ist es für Macintosh-Programme. Es ist sehr wichtig, daß sie jedes Ereignis mitbekommen und möglichst schnell darauf reagieren. Die schnelle Reaktion auf alle Ereignisse ist (hoffentlich) eine direkte Folge der typischen Struktur eines Macintosh-Programms. Ereignisse werden an zentraler Stelle registriert, eingeordnet und weitergeleitet.

Es ist aber gar nicht so einfach, "auf dem laufenden zu bleiben". Selbst wenn sich ein Programm in einer Berechnung befindet, die vielleicht 1 oder 2 Sekunden dauert, dürfen trotzdem keine Tasten-Drücke verlorengehen, die in dieser Zeit stattfanden. Dies läßt sich normalerweise nur durch recht komplexe (interrupt-gesteuerte) Programmstrukturen erreichen. Zudem benötigt das Erkennen und Klassifizieren von Ereignissen detaillierte

Hardwarekenntnisse (welches Bit an welcher Speicherstelle ändert sich von 0 auf 1, wenn eine bestimmte Taste gedrückt wird ?). Diese ganze Arbeit nimmt einem der EventManager ab.

Der EventManager arbeitet mit dem Betriebssystem des Macintosh zusammen und sorgt (interrupt-gesteuert) dafür, daß alle Ereignisse registriert werden, auch wenn gerade eine lange Berechnung vom Anwendungsprogramm durchgeführt wird. Nach dem Erkennen eines Ereignisses von einem bestimmten Typ wird ein entsprechender EventRecord erzeugt und dessen restliche Felder (**where, modifiers** etc.) entsprechend gefüllt. Dieser EventRecord wird dann an das Ende einer Warteschlange gesetzt, in der sich vielleicht schon andere Ereignisse befinden, die das Anwendungsprogramm noch nicht bearbeitet hat.

Hat das gerade aktive Anwendungsprogramm den letzten Event komplett bearbeitet, so holt es sich aus dieser Warteschlange den nächsten EventRecord und bearbeitet diesen. Alle anderen Events rücken dann um eine Stelle in der Schlange auf — wie es sich für eine ordentliche Warteschlange gehört. Da der EventManager alle komplizierten Buchhaltungsaufgaben erledigt, die mit dem Erkennen und "Lagern" von Ereignissen zu tun haben, braucht sich das Programm nur noch um ihre Abarbeitung zu kümmern.

6.5 Die Funktionen des EventManagers

Nach all den erklärenden Worten über das Wie und Warum des Ereignis-Konzeptes auf dem Macintosh sollen nun die Funktionen erläutert werden, über die ein Anwendungsprogramm mit dem EventManager zusammenarbeitet — es sind nur einige wenige:

```
FUNCTION GetNextEvent
                  (eventMask:    INTEGER;
                   VAR theEvent: EventRecord
                  ): BOOLEAN;
```

GetNextEvent ist die wichtigste Funktion des ganzen EventManagers. Sie stellt fest, ob irgendwelche Ereignisse in der Warteschlange liegen, und liefert das erste davon im VAR-Parameter **theEvent** zurück und entfernt es aus der Schlange. Liegt kein Ereignis in der Warteschlange, wird ein **nullEvt** zurückgegeben. Diese Funktion wird stets am Beginn der Haupt-Schleife im Programm aufgerufen (mehr dazu im nächsten Kapitel). Der Aufbau eines

EventRecords und die darin enthaltenen Informationen wurden bereits weiter oben erläutert.

Im Parameter **eventMask** kann man (bitweise verschlüsselt) angeben, welche Ereignisse berücksichtigt werden sollen. Jedes der 16 Bits von **eventMask** entspricht dabei einem der 16 Ereignis-Typen (Bit 1 z.B. einem **mouseDown**). Ist das entsprechende Bit gleich 1, so werden Ereignisse dieses Typs berücksichtigt. Ist das Bit gleich 0, so werden Ereignisse dieses Typs nicht an das Programm zurückgegeben, selbst wenn ein solcher Event als nächster in der Schlange wartet. Üblicherweise sollte man für **eventMask** immer die Konstante **everyEvent** (=1111111111111111 in binärer Schreibweise) übergeben, damit alle Events an das Programm gemeldet werden können. Dies ist selbst dann sinnvoll, wenn man sich um einige der Event-Typen nicht kümmern will, damit diese aus der Warteschlange entfernt werden und keinen Platz kosten.

Die Funktion **GetNextEvent** liefert den Wert TRUE, wenn es sich bei dem Event **theEvent** um ein Ereignis handelt, um das sich das Anwendungs - programm kümmern muß. Eine Reihe von Ereignissen werden von **GetNextEvent** direkt an andere Teile der ToolBox weitergereicht und brauchen dann vom Programm nicht mehr bearbeitet zu werden. Sie werden "für alle Fälle" aber trotzdem noch in **theEvent** zurückgegeben, wobei **GetNextEvent** aber FALSE ergibt.

```
FUNCTION EventAvail
                (eventMask:     INTEGER;
                 VAR theEvent:  EventRecord
                ): BOOLEAN;
```

EventAvail ist nahezu hundertprozentig identisch mit **GetNextEvent**. Nur wird das Ereignis, das in **theEvent** zurückgegeben wird, nicht aus der Warteschlange entfernt. **EventAvail** ist deshalb eine Methode, um nachzuschauen, welche Events sich noch in der Warteschlange befinden, ohne deren Inhalt zu beeinflussen. Sie wird aber von den meisten Programmen nie oder nur sehr selten benötigt.

Die folgenden Funktionen dienen nur der Abfrage der Maus und sind besonders für Programmschleifen wichtig. Viele Schleifen in Grafikprogrammen bewegen z.B. irgend etwas auf dem Bildschirm und folgen dabei der Maus so lange, bis entweder die Maus-Taste gedrückt oder gelöst wird.

```
PROCEDURE GetMouse (VAR mouseLoc: Point);
```

GetMouse gibt im VAR-Parameter **mouseLoc** die aktuelle Position des Maus-Cursors zurück. Die Koordinaten von **mouseLoc** sind dabei im aktuellen GrafPort (als lokal) aufzufassen, der meist dem obersten Fenster entsprechen dürfte.

```
FUNCTION Button: BOOLEAN;
```

Die Funktion **Button** liefert TRUE, falls die Maus-Taste gerade gedrückt ist, und FALSE sonst.

GetMouse und **Button** haben beide eigentlich nicht viel mit Ereignissen zu tun, sondern fragen nur — unabhängig vom Eintreten eines Ereignisses — den aktuellen Zustand der Maus-Hardware/Software ab. Die beiden folgenden Funktionen allerdings haben etwas mit Ereignissen zu tun und sind für die meisten Anwendungen einem Aufruf von **Button** vorzuziehen.

```
FUNCTION StillDown: BOOLEAN;
```

Die Funktion **StillDown** entspricht im wesentlichen der Funktion **Button**. Sie liefert so z.B. auch FALSE, wenn die Maus-Taste gelöst ist. Ist die Maus-Taste aber gedrückt, so liefert die **StillDown** nur dann den Wert TRUE, wenn sie zwischen dem letzten von **GetNextEvent** gelieferten **MouseDown**-Event und dem Augenblick des Aufrufs von **StillDown** nie gelöst wurde. Lag zwischen dem zuletzt bearbeiteten **MouseDown** und dem Aufruf von **StillDown** ein zweiter **MouseDown**, so liefert sie FALSE.

```
FUNCTION WaitMouseUp: BOOLEAN;
```

WaitMouseUp arbeitet ähnlich wie **StillDown**, nur entfernt sie zusätzlich einen eventuell in der Warteschlange liegenden **MouseUp**-Event aus dieser, bevor sie mit dem Ergebnis FALSE zurückkehrt. Kehrt sie mit dem Ergebnis TRUE zurück, ist ihr Verhalten identisch mit **StillDown**. Dies ist aber nur für Programme interessant, die sich überhaupt um **MouseUp**-Events kümmern und gleichzeitig Doppelklicks eine besondere Bedeutung beimessen – und das sind nur wenige.

6.6 Schlußbemerkung

Das waren dann aber schon alle Funktionen des EventManagers, mit denen ein Programm normalerweise zu tun hat. Sie werden üblicherweise auch nur im Programm-Rahmen benötigt, wo die Ereignisse verteilt werden. Innerhalb der Erläuterung dieses Rahmenprogramns – im zweiten Teil dieses Buches – werden diese Operationen noch einmal ausführlich am Beispiel erläutert.

Steht dieser Rahmen erst mal, braucht man sich nur selten um den EventManager zu kümmern (es kann höchstens mal ein gelegentlicher Aufruf von **GetMouse** und **StillDown** nötig werden). Trotzdem sollte man sich vor dem Lesen des nächsten Kapitels die Arbeitsweise des EventManagers noch einmal genau anschauen und versuchen ihn zu verstehen, damit man weiß, weshalb der Rahmen so und nicht anders aussieht.

7 Das Rahmenprogramm

Nach der 6 Kapitel langen Vorrede nehme ich an, daß vielleicht der eine oder andere Leser etwas ungeduldig geworden ist und nun annimmt, daß es sich bei diesem Buch um eine Kurzform des Macintosh-Programmier-Handbuchs handelt. Dem ist nicht so! Wir gelangen nun zum wirklichen Programmieren. Beginnend mit diesem Kapitel soll das schon mehrfach angesprochene Rahmenprogramm entwickelt werden, anhand dessen ich die Regeln für das Programmieren des Macintosh verdeutlichen will. Die Hauptleistung dieses Rahmenprogramms wird es sein, die Ereignisse (*Events*), die vom Benutzer und der Macintosh-Fensterverwaltung kommen, korrekt zu empfangen, zu entschlüsseln und (vor allem!) korrekt zu bearbeiten, wie es von einem "richtigen" Macintosh-Programm erwartet wird.

Der Rest dieses Kapitels besteht aus Stücken von Programmen (Prozeduren und Funktionen) und den zugehörigen Erläuterungen im schnellen Wechsel. Die Folge der Programmstücke ist im wesentlichen nach dem TopDown-Prinzip angeordnet. D.h. zunächst werden immer die Programmteile beschrieben, die sich auf einer "höheren Ebene" bewegen, dann kommen die Beschreibungen der von diesen Programmteilen verwendeten Prozeduren. Gehören mehrere Prozeduren und Funktionen eng zusammen, so stehen sie auch im Text direkt hintereinander, wie später im fertigen Programm. Bei diesen Gruppen von Prozeduren handelt es sich meist um eine Haupt-Prozedur und einige Hilfsroutinen. Sie werden – im Gegensatz zum Gesamtaufbau des Kapitels – in der Reihenfolge aufgelistet, wie sie auch der Compiler erwartet: jede Prozedur oder Funktion wird zunächst definiert, bevor sie verwendet wird.

Im folgenden Abschnitt soll zunächst kurz beschrieben werden, was unser Beispiel-Programm überhaupt leisten soll, um einen Eindruck davon zu bekommen, vor welcher Aufgabe wir stehen.

7.1 Was leistet das Beipielprogramm?

Die in der Überschrift gestellte Frage läßt sich kurz mit "Eigentlich nicht viel." beantworten. Man muß allerdings dazusetzen: "Aber das richtig!" Unser erstes Beipielprogramm soll ja nur zur Demonstration des Rahmenprogramms dienen, und da wäre es nur unübersichtlich, es durch viele Funktionen zu überladen. Das Programm wird deshalb in etwa die folgenden Aufgabenbereiche eines Macintosh-Programms abdecken:

- Menüs und Auswahl aus Menüs mit folgenden Möglichkeiten
  - Graue Menü-Punkte abhängig vom obenliegenden Fenster
  - Tastatur-Äquivalente für Menü-Befehle
- 2 überlappende Fenster mit folgenden Möglichkeiten:
  - Öffnen und Schließen
  - Verschieben
  - Nach-vorne-Holen und Nach-hinten-Legen
- Zeichnen zweier einfacher Grafiken in den beiden Fenstern
- Bewegen der Grafiken mittels der Maus

Die Fenster werden Standard-Dokumenten-Fenster sein, wie sie z.B. auch MacWrite verwendet. Diese Fenster haben eine Titel-Leiste mit ihrem Namen und einem kleinen Kästchen auf der linken Seite dieser Leiste, mit dem sie geschlossen werden können. Es werden die folgenden Menüs und Menü-Punkte realisiert:

- Ablage
  - *Öffne Fenster 1* (ist grau, wenn das Fenster offen ist)
  - *Öffne Fenster 2* (ist grau, wenn das Fenster offen ist)
  - *Beenden* (mit dem Tastatur-Äquivalent 'Q')
    Beenden ist durch einen grauen Strich
    von den anderen Punkten getrennt
- Bearbeiten
  - *Größer* (mit dem Tastatur-Äquivalent 'G')
  - *Kleiner* (mit dem Tastatur-Äquivalent 'K')
  - *1 nach oben* (ist grau, wenn Fenster 1 oben liegt)
  - *2 nach oben* (ist grau, wenn Fenster 2 oben liegt)

Die folgende Abbildung zeigt das "typische" Aussehen des Bildschirms während des Programm-Ablaufs:

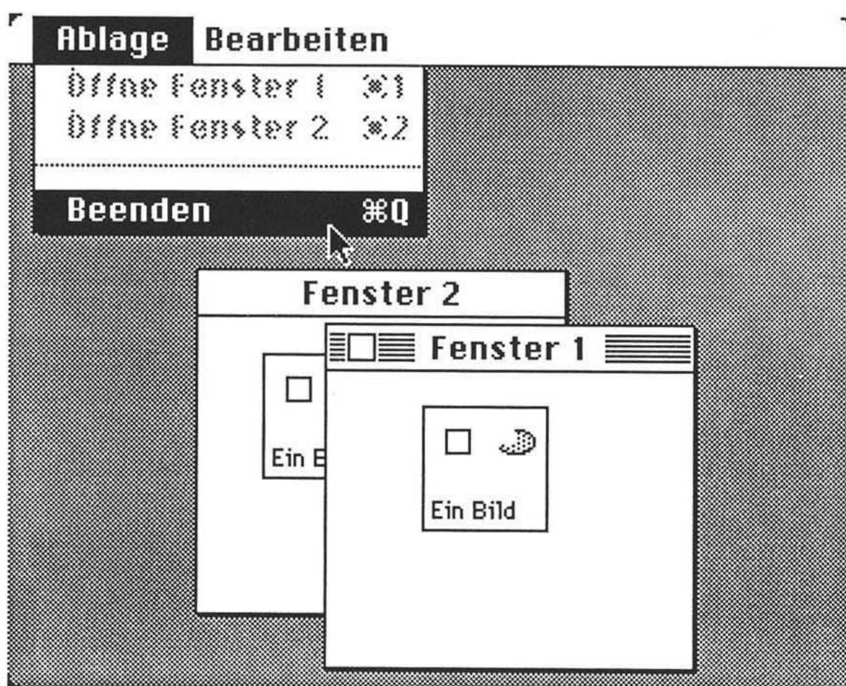


Bild 7 - 1: *Das Beispiel-Programm in Aktion*

7.2 Das Hauptprogramm

Zunächst soll das Hauptprogramm des Beipiel-Programms vorgestellt werden. Es sollte im wesentlichen (in etwas anderer Schreibweise) bereits aus dem Kapitel *Ereignisse* bekannt sein. Ich verwende, wie schon im ersten Teil dieses Buches, für alle Beispiele die Programmiersprache Pascal. Eine Übertragung in einen besonderen Dialekt wie MacAdvantage™, C oder Modula sollte aber niemandem, der in diesen Sprachen einigermaßen bewandert ist, Schwierigkeiten machen.

Das Hauptprogramm eines Macintosh-Programms ist typischerweise sehr klein und besteht nur aus wenigen Zeilen. Es läßt sich in der hier vorgestellten Form eigentlich problemlos für jede Art von Programm verwenden:

```
VAR
        ende:       BOOLEAN;

BEGIN {HauptProgramm}
        InitMac;
        InitProg;
        ende := false;
        REPEAT
            BearbeiteEreig;
        UNTIL ende;
        BeendeProgramm;
END.{HauptProgramm}
```

Kern des Hauptprogramms ist die sogenannte (Haupt-)Event-Schleife (auf engl. *Main Event Loop*; oder kurz MEL genannt), in der die von außen kommenden Ereignisse vom Betriebssystem erfragt und dann bearbeitet werden (in **BearbeiteEreig**). Kontrolliert wird die Schleife durch eine BOOLEsche Variable namens **ende**, die dann TRUE wird, wenn das Ende des Programms vom Benutzer gewünscht wird (durch einen Menü-Befehl). Der Event-Schleife voran geht ein Initialisierungs-Teil, in dem zunächst die nötigen Initialisierungen für das Macintosh-Betriebssystem vorgenommen werden (**InitMac**) und danach alle für dieses konkrete Programm benötigten Initialisierungen (**InitProg**). In der Programmiersprache Pascal ist es im allgemeinen z.B. nötig, Variablen explizit mit einem Start-Wert zu belegen und dynamisch erzeugte Datenstrukturen anzulegen. Am Ende des Programms erledigt ein Terminierungs-Teil eventuell benötigte "Aufräum-Arbeiten". Das Macintosh-Betriebssystem benötigt bei der Beendigung eines Programms keinerlei Aufräum-Arbeiten; diese könnten höchstens für das Anwendungsprogramm anfallen (z.B. schließen offener Fenster und Dateien). In unserem ersten Beispiel-Programm allerdings wird **BeendeProgramm** noch leer bleiben.

7.3 Die Initialisierungsphase

Bevor ein Programm auf dem Macintosh wirklich mit seiner eigentlichen Arbeit beginnen kann, müssen verschiedene Initialisierungsaufgaben durchgeführt werden. Diese Aufgaben zerfallen in zwei Teile. Erst müssen die verschiedenen Komponenten des Betriebssytems initialisiert werden und eventuell für dieses spezielle Programm konfiguriert werden. Danach muß aber auch das Programm selbst auf die Benutzung vorbereitet werden. Hierzu gehört z.B. das Initialisieren von Variablen, aber auch das Anlegen von

Menüs und Fenstern, die der Benutzer bereits beim Start des Programms sehen soll.

7.3.1 Initialisierung des Betriebssystems

Die Initialisierung der einzelnen Komponenten des Macintosh-Betriebssystems kann in standardisierter Form geschehen und von jedem Programm übernommen werden, das in derselben Programmiersprache geschrieben wird. Zwischen den Programmiersprachen allerdings bestehen teilweise erhebliche Unterschiede. In manchen Sprachen (z.B. LisaPascal) muß das eigentliche Anwendungsprogramm sämtliche Initialisierungs-aufgaben selbst erledigen. Hingegen werden in anderen Sprachen einige der wichtigsten Initialisierungen bereits von "unsichtbaren" Programmteilen ausgeführt, bevor die erste Instruktion des Hauptprogramms beginnt. Dies ist z.B. bei MacAdvantage™ UCSD-Pascal und einigen C-Compilern der Fall.

Welche Initialisierungen "die Sprache selbst" vornimmt und welche für das Anwendungsprogramm übrigbleiben, entnimmt der Leser am besten der Dokumentation für die Programmiersprache, mit der er arbeitet. Ich stelle die Prozedur **InitMac** in ihrer allgemeinsten Form vor. Streichungen möge der Leser bitte anhand der entsprechenden Dokumentation vornehmen.

```
VAR
     dragRect: Rect;

PROCEDURE InitMac;
BEGIN
     InitGraf(@thePort); { in LisaPascal }
     { InitGraf(&thePort); in C }
     InitFonts;
     InitWindows;
     FlushEvents(everyEvent,0);
     TEInit;
     InitDialogs(NIL);

     dragRect := screenBits.bounds;
     dragRect.top := dragRect.top + 20;
                    { Menü-Leiste oben abziehen }
     InsetRect(dragRect,4,4);
                    { ringsherum 4 Pixel kleiner}

     InitCursor;
END;{InitMac}
```

Am Anfang wird mit **InitGraf** das Grafikpaket QuickDraw initialisiert und bekommt Speicherplatz für seine eigenen Variablen (bei **thePort**) zugewiesen. **ThePort** ist in den meisten Programmiersprachen eine vordefinierte globale Variable, deren Adresse an **InitGraf** übergeben werden muß. Diese Adresse wird in LisaPascal normalerweise durch den Operator @ und in C durch den Operator & ermittelt.

InitFonts initialisiert den FontManager, der für die Verwaltung der vielen unterschiedlichen Zeichensätze, die QuickDraw verwenden kann, zuständig ist. **InitWindows** initialisiert in entsprechender Weise den WindowManager. Beide Aufrufe reservieren dynamisch Speicherplatz auf dem Heap zur eigenen Verwendung.

FlushEvents entfernt bei einem Aufruf in der vorgestellten Form alle Ereignisse, die sich noch in der Warteschlange befinden, daraus. Dies ist nötig, damit nicht ein Tastendruck oder ein Maus-Klick, den man gemacht hat, bevor das Programm gestartet war (während man sich noch im Finder befand), von diesem Programm verarbeitet wird. Die meisten Macintosh-Programme rufen **FlushEvents** zu Beginn des Programmlaufs auf. Wünscht man statt dessen, daß Events, die noch vor dem Start des Programms passiert sind, vom Programm bearbeitet werden sollen, läßt man

FlushEvents einfach fort. Dies ist vielleicht bei kleineren Programmen ganz sinnvoll, damit man diesen bereits Befehle geben kann, während der Finder sie lädt.

TEInit initialisiert das **TextEdit**-Paket in der ToolBox, das immer dann verwendet wird, wenn irgendeine textuelle Eingabe vom Benutzer verlangt wird. Das TextEdit-Paket wird in einem späteren Kapitel noch ausführlicher behandelt. Aber selbst wenn ein Programm es nicht explizit verwendet, sollte in der Initialisierungsphase **TEInit** aufgerufen werden, da sich andere Teile der ToolBox darauf verlassen, daß TextEdit initialisiert ist, wenn sie aktiv werden.

InitDialogs initialisiert den DialogManager, den wir ebenfalls im einem folgenden Kapitel noch näher behandeln werden. Der DialogManager sollte übrigens auch dann initialisiert werden, wenn man ihn selbst nicht explizit aufruft, da sich andere Teile der ToolBox darauf verlassen, daß er initialisiert ist, wenn sie aktiv werden. Üblicherweise sollte beim Aufruf von **InitDialogs** immer der Parameter NIL übergeben werden. Nur sehr fortgeschrittene Programmierer sollten hier die Adresse einer Prozedur übergeben, die dann aufgerufen werden soll, wenn ein schwerwiegender Systemfehler (Bombe) passiert ist und der Benutzer auf den Knopf namens *Retten* im entsprechenden Fenster drückt.

DragRect ist ein Rechteck, das später die Bewegung eines Fensters begrenzt. Es wird an der entsprechenden Stelle als Parameter für die Prozedur **DragWindow** im WindowManager verwendet. **DragRect** orientiert sich am Rechteck **screenBits.bounds**, den Grenzen der BitMap, die den Bildschirmpuffer des Macintosh darstellt. **ScreenBits** steht in den meisten Programmiersprachen als globale Variable von QuickDraw zur Verfügung. Indem wir **dragRect** von **screenBits.bounds** abhängig machen, erreichen wir, daß unser Programm problemlos auch auf der Lisa und anderen Macintosh-kompatiblen Rechnern, die eventuell andere Bildschirm-Abmessungen haben, läuft.

InitCursor schließlich setzt den Cursor explizit auf seine "Normalform"; einen nach links oben gerichteten Pfeil. Dieser Aufruf ist deshalb nötig, da der Finder, wenn er ein Programm startet, den Cursor immer auf das Uhr-Symbol setzt, um dem Benutzer anzudeuten, daß ein Vorgang abläuft, der etwas länger dauern kann.

Alle hier beschriebenen Initialisierungen sollten unbedingt in der vorgestellten Reihenfolge durchgeführt werden! Teilweise verläßt sich ein ToolBox-Aufruf

auf die Initialisierung anderer Teile der ToolBox. Abweichungen von der Reihenfolge in InitProg sind deshalb riskant und sollten wohlüberlegt sein.

7.3.2 Initialisierung des Programms

Die Initialisierung des eigentlichen Anwendungsprogramms (**InitProg**) ist im Gegensatz zu **InitMac** kaum von einem Programm zum anderen übertragbar. Der grundsätzliche Aufbau dieser Prozedur ist allerdings von Programm zu Programm weitgehend gleich. Ich liste nun die Prozedur **InitProg** auf, die diese Initialisierung für unser einfaches Beispielprogramm macht. Zunächst kommen die globalen Variablen des Programms, die von dieser Prozedur initialisiert werden, dann die Prozedur selbst.

```
CONST
    lastMenu=        2;

    mAblage=         1;
    mBearbeiten=     2;

VAR
    menu:        ARRAY[1..lastMenu]OF MenuHandle;
    fenster1:    WindowPtr;
    fenster2:    WindowPtr;
    wPict:       PicHandle;
    scale:       INTEGER;


FUNCTION  CreatePict: PicHandle;
BEGIN
     ...... { Wird weiter unten beschrieben }
END;{CreatePict}
```

```
PROCEDURE InitProg;
VAR
    i:                  INTEGER;
BEGIN
    MoreMasters;
    MoreMasters;
    MoreMasters;
    MoreMasters;

    menu[mAbl]
        := NewMenu(mAblage,'Ablage');
    menu[mBearbeiten]
        := NewMenu(mBearbeiten,'Bearbeiten');
    AppendMenu(menu[1],
            '(Öffne Fenster 1/1;
            Öffne Fenster 2/2;
            (-;
            Beenden/Q');
    AppendMenu(menu[2],
            'Größer;
            Kleiner;
            (-;
            1 nach oben;
            (2 nach oben');
    FOR i:= 1 TO lastmenu DO
        InsertMenu(menu[i],0);
    DrawMenuBar;
```

```
        SetRect(bounds,100,100,300,300);
        fenster1 := NewWindow(NIL,bounds,
                    'Fenster 1', { Titel }
                    TRUE,        { sichtbar !  }
                    documentProc,
                    Ptr(-1),     { oberstes Fenster}
                    TRUE,        { Schließ-Knopf }
                    0);          { refCon = 0 }
        SetWKind(fenster1,grafKind);

        SetRect(bounds,110,110,310,310);
        fenster2 := NewWindow(0,bounds,
                    'Fenster 2',
                    FALSE,       { UNsichtbar !    }
                    documentProc,
                    Ptr(-1),
                    TRUE,0);
        SetWKind(fenster2,textKind);

        SelectWindow(fenster1);

        wPict := CreatePict;
        scale := 8;

    END;{InitProg}
```

InitProg ruft zunächst viermal die Prozedur **MoreMasters** auf, die jeweils einen Block von 64 *MasterPointern* für die Verwaltung von Handles neu anlegt. Die MasterPointer sind logischerweise nicht verschiebbare Blöcke im Heap, da auf sie ja mit direkten Zeigern verwiesen wird. Am Anfang des Programms sollten deshalb so viele wie davon nötig – lieber ein paar mehr – angelegt werden. Jeder später noch neu angelegte MasterPointer stellt eine Insel im Heap dar, die eine mögliche Kompaktierung stören würde. Wer dabei im Moment Verständnisschwierigkeiten hat, sollte noch einmal die Beschreibung von Handles im Kapitel Speicherverwaltung nachlesen!

Danach wird die Menü-Leiste zusammengestellt und mit **DrawMenuBar** am Bildschirm gezeigt. Die mittels **NewMenu** erzeugten und mit **AppendMenu** vervollständigten MenuHandles werden vorher im Array **menu** gespeichert. Da das Resource-Konzept des Macintosh erst in einem späteren Kapitel besprochen wird, stehen die Texte – sowohl für die Menü-Titel wie auch für ihre Punkte – direkt im Programm-Text. Dies ist aber nicht

der "Normalfall" in einem Macintosh-Programm, wie wir später noch sehen werden.

Danach werden zwei Fenster erzeugt, von denen eins zunächst unsichtbar bleibt. Zeiger auf diese Fenster werden in den globalen Variablen **fenster1** und **fenster2** gespeichert. Das Erzeugen der Fenster geschieht mit zwei Aufrufen von **NewWindow**, denen alle benötigten Parameter direkt übergeben werden. Die Abmessungen und die Position der beiden Fenster wird vorher im Rechteck **bounds** gespeichert. Durch Übergabe eines NIL als ersten Parameter wird der **WindowManager** angewiesen, den benötigten Speicherplatz für dieses Fenster selbst im Heap anzulegen. Die anderen Parameter werden alle direkt im Listing erläutert. Nachdem ein Fenster erzeugt wurde, wird (bereits im Hinblick auf spätere Erweiterungen) mit der Prozedur **SetWKind** das Feld **windowKind** des entsprechenden WindowRecords auf die Konstante **grafKind** (für GrafikFenster) gesetzt. Die Prozedur **SetWKind** muß man sich für Pascal selbst schreiben (sie ist weiter unten erläutert), in C oder Assembler kann man das entsprechende Feld des WindowRecords direkt setzen.

Bei Verständnisproblemen zu diesem Abschnitt sollte der Leser am besten noch einmal die Kapitel *Fenster-Verwaltung* und *QuickDraw* durchlesen!

Nachdem die beiden Fenster erzeugt wurden, wird das erste durch **SelectWindow** zum obersten gemacht und mit **SetPort** zum aktuellen GrafPort. Zum Abschluß von **InitProg** wird noch das Zeichnen einer Grafik in den beiden Fenstern vorbereitet. Diese Grafik wird in Form eines QuickDraw-Pictures in der globalen Variablen **wPict** gespeichert. Sie wird durch die Funktion **CreatePict** erzeugt, die uns ja bereits aus dem Beispielprogramm am Ende des Kapitels *QuickDraw* bekannt ist. Wir werden in diesem Programm aber (über zwei Menübefehle) die Möglichkeit anbieten, die Grafik größer oder kleiner zu machen. Hierzu wird uns der Faktor **scale** dienen, den wir hier mit 8 vorbesetzen. Die Abmessungen des Bildes **wPict** werden später beim Zeichnen mit den Faktor (**scale** DIV 8) multipliziert werden.

```
FUNCTION  CreatePict: PicHandle;
VAR
    pict:               PicHandle;
    saveClip:           RgnHandle;
    r:                  Rect;
BEGIN
    SetRect(r,1,1,50,50);
    saveClip := NewRgn;
    GetClip(saveClip);
    ClipRect(r);
    pict := OpenPicture(r);
        PenSize(1,1);
        PenPat(black);
        PenMode(patXor);
        TextSize(9);
        TextMode(patXor);
        TextFond(geneva);
        FrameRect(r);
        SetRect(r,10,10,20,20);
        FrameRect(r);
        PenPat(dkGray);
        PenSize(2,2);
        SetRect(30,10,45,20);
        FramArc(r,0,250);
        PenPat(ltGray);
        PaintArc(r,0,250);
        MoveTo(5,45);
        DrawString('Ein Bild');
    ClosePicture;
    CreatePict := pict;
    SetClip(saveClip);
    DisposeRgn(saveClip);
END;{CreatePict}
```

7.4 Event-Verarbeitung

Der wichtigste Teil des Hauptprogramms besteht aus der Event-Schleife, in der die von außen kommenden Ereignisse aus der vom Betriebssystem verwalteten Warteschlange entnommen werden und je nach Art des Ereignisses weiterverarbeitet werden. Wie die Ereignisse in die

Warteschlange geraten, interessiert dabei überhaupt nicht. Wir empfangen und bearbeiten sie nur.

```
REPEAT
     BearbeiteEreig;
UNTIL ende;
```

Die Eventschleife besteht in der hier vorgestellten Form nur aus einem Prozedur-Aufruf **BearbeiteEreig**. **BearbeiteEreig** entfernt das erste Ereignis aus der Warteschlange und führt eine Vorverarbeitung damit durch. Außerdem werden hier periodische Aktivitäten durchgeführt, die während des Programmlaufs immer passieren müssen. (Zu diesen periodischen Aktivitäten gehören z.B. das Ändern der Cursor-Form je nach dem Gebiet, über dem sich der Cursor gerade befindet und das Blinken von Einfügestellen in Text etc.) Unser erstes Beipielprogramm wird allerdings noch keine solchen Aktivitäten durchführen. Die wesentliche Funktion von **BearbeiteEreig** ist die Analyse des aktuellen Ereignisses und darauffolgend die Weitergabe der Kontrolle an die Prozeduren, die die "wirkliche Arbeit" leisten. Diese Prozedur trägt also nicht die Hauptlast der Ereignis-Verarbeitung, sondern "delegiert" im wesentlichen nur.

7.4.1 Die Prozedur BearbeiteEreig

```
VAR          { globale Variablen für die
               entsprechenden
               Bits in event.modifiers }
   fMouse,
   fShift,
   fShLock,
   fOption,
   fCommand,
   fActive:     BOOLEAN;

   dasEreignis: EventRecord;

PROCEDURE BearbeiteEreig;
VAR
   ch:     CHAR;
   menuErg: LONGINT;
BEGIN
   {   Hier stehen bei einem wirklichen
       Anwendungsprogram Prozedur-Aufrufe für alle
       periodischen Aktivitäten eines Programms
   }
   IF GetNextEvent(everyEvent,dasEreignis) THEN
   WITH dasEreignis DO
       BEGIN
       fMouse   := (BitAnd(modifiers,128) <> 0);
       fShift   := (BitAnd(modifiers,512) <> 0);
       fShLock  := (BitAnd(modifiers,1024)<> 0);
       fOption  := (BitAnd(modifiers,2048)<> 0);
       fCommand := (BitAnd(modifiers,256) <> 0);
       fActive  := (BitAnd(modifiers,001) <> 0);
```

```
        CASE what OF
            mouseDown:
                MausKlick(where);
            autoKey,keyDown:
                BEGIN
                ch := CHR(BitAnd(message,255));
                IF   fCommand
                AND (what = keyDown) THEN
                    BEGIN
                    menuErg := MenuKey(ch);
                    DoCommand( HiWord(menuErg),
                               LoWord(MenuErg));
                    END;
                END;
            updateEvt:
                WindowUpdate(WindowPtr(message));
            activateEvt:
                WindowActivate(WindowPtr(message),
                               fActive);

            {   Alle anderen Event-Arten werden
                (im Moment) noch nicht bearbeitet
            }
        END;{CASE};
        END;
END;{BearbeiteEreig}
```

BearbeiteEreig entfernt zunächst einen Event mit **GetNextEvent** aus der Warteschlange und analysiert ihn dann. Er liegt nach dem Aufruf von **GetNextEvent** in der globalen Variablen **dasEreignis** und kann von jeder Prozedur des Programms aus verwendet werden. Ergibt der Aufruf von **GetNextEvent** FALSE, d.h. es ist kein Ereignis in der Warteschlange, das das Programm bearbeiten könnte, tut **BearbeiteEreig** gar nichts.

Ergibt **GetNextEvent** TRUE, werden als erstes die Bits des **modifiers**-Feldes dieses Events analysiert. Es handelt sich dabei ja eigentlich um BOOLEsche Variablen, die Aussagen über den Zustand der Tastatur und der Maus zum Zeitpunkt, als der Event eintrat, enthalten. Da in Pascal die Abfrage einzelner Bits nur umständlich möglich ist, "entpacken" wir diese Bits zu echten BOOLEschen Variablen. Die globalen Variablen **fMouse**, **fShift**, **fShLock**, **fOption**, **fCommand** und **fActive** enthalten die Werte der entsprechenden Bits nach dem Aufruf von **BearbeiteEreig** in etwas leichter zu handhabender Form. Die Funktion **BitAnd** aus der ToolBox, mit

der die einzelnen Bits von **modifiers** geprüft werden, realisiert eine bitweise logische UND-Operation, die es in dieser Form in der Sprache Pascal nicht gibt (in C ist sie z.B. eingebaut).

Nachdem diese Flags extrahiert und in eine etwas lesbarere Form gebracht wurde, wird das Feld **what** des EventRecord inspiziert. Der Wert von **what** unterscheidet die 16 möglichen Arten von Events voneinander. Wir bearbeiten nur 4 dieser 16 möglichen Ereignis-Typen. Die anderen müssen nur von den wenigsten Programmen oder nur sehr selten bearbeitet werden.

Mit die wichtigsten und komplexesten Ereignisse sind Maus-Klicks. Diese reichen wir an die Prozedur **MausKlick** weiter, die weiter unten näher beschrieben wird.

Auf Tastendrücke wird unser erstes Programm nur dann reagieren, wenn es sich um ein Menü-Äquivalent handelt (also die Befehls-Taste gleichzeitig gedrückt ist). In diesem Fall entnehmen wir den gedrückten Buchstaben dem Wert des Feldes **message** und stellen über die Funktion **MenuKey** fest, welchem Menü-Befehl diese Tasten-Kombination entspricht. Wie bereits im Kapitel über den MenuManager beschrieben, wird danach das Ergebnis von **MenuKey** in die Kenn-Zahl (ID) des gewählten Menüs und die Position des gewählten Menü-Punktes zerlegt. Was auf die Auswahl eines bestimmten Menü-Punktes hin passiert, ist sehr programmspezifisch und kaum in allgemeiner Form zu programmieren. Diese Weiterverarbeitung übernimmt die Prozedur **DoCommand**, die weiter unten noch näher beschrieben wird.

Die nächste Gruppe von Ereignissen, auf die jedes Programm reagieren muß, sind die Update-Events. Ein Update-Event signalisiert immer, daß ein Fenster-Inhalt (bzw. ein Teil davon) neugezeichnet werden muß. Diese Aufgabe überlassen wird der Prozedur **WindowUpdate**, die aus dem ihr übergebenen Parameter entnehmen kann, welches Fenster neu gezeichnet werden muß. Dieser Parameter ist der Wert des Feldes **message** des EventRecords, in dem sich bei **updateEvts** und **activateEvts** das betroffene Fenster verbirgt.

Genauso reichen wir einen **activateEvt** an die Prozedur **WindowActivate** weiter, der wir in einem zweiten Parameter noch mitteilen, ob das betroffene Fenster aktiv wird (nach oben kommt) oder inaktiv wird (nach unten kommt oder geschlossen wird). **WindowActivate** unternimmt in den meisten Programmen nicht viel und aktiviert oder deaktiviert typischerweise nur ein paar Menü-Befehle, die nur in Abhängigkeit von einem bestimmten oben liegenden Fenster ausgewählt werden können.

Die Prozeduren, an die wir **updateEvts** und **acvtivateEvts** weiterleiten, erwarten beide als Parameter einen **WindowPtr**. Das Feld **message** des EventRecords enthält auch einen solchen **WindowPtr**; deklariert wird es jedoch als Feld vom Typ LONGINT. Damit wir seinen Inhalt an die Prozeduren übergeben können, müssen wir den Typschutz, der in den meisten höheren Programmiersprachen (glücklicherweise) eingebaut ist, umgehen. In LisaPascal oder Modula2 sieht das aus, wie ich es im Listing gezeigt habe. In C und MacAdvantage ist eine Umgehung des Typschutzes aus unterschiedlichen Gründen nicht nötig.

Das einzige Ereignis, das wir zwar schon empfangen, aber noch nicht richtig bearbeiten, ist der **autoKey**-Event. Er entspricht einem Tastendruck, der aber nicht vom Benutzer kommt, sondern durch die Auto-Repeat-Funktion der Tastatur ausgelöst wurde. Er wird zunächst genauso analysiert wie der **keyDown**-Event, d.h. es wird festgestellt, welche Taste gedrückt wurde. Da wir jedoch nur auf Tastatur-Äquivalente von Menü-Befehlen reagieren und diese im allgemeinen nicht mit einem durch die Repeat-Funktion hervorgerufenen Tastendruck ausgelöst werden können, wird das Ereignis ignoriert. Sobald unser Programm auf Text-Eingaben reagieren soll, müssen wir diesen "synthetisch" erzeugten Tastendruck (genauso wie auch die normalen Tastendrücke mit nicht gedrückter Befehls-Taste) an eine entsprechende Prozedur weiterleiten.

Das hier beschriebene Aussehen der Prozedur **BearbeiteEreig** ist bis jetzt noch völlig allgemeingültig und ist auf nahezu jedes Programm übertragbar. Die programmspezifische Weiterverarbeitung geschieht erst in den von ihr aufgerufenen Prozeduren (wie z.B. **DoCommand**). Mit ihnen wollen wir uns im folgenden beschäftigen. Auch in diesen finden sich noch viele übertragbare Teile, ihre Aufgaben sind zum großen Teil aber abhängig vom konkreten Programm. Ich werde dabei vor allem versuchen, diejenigen Teile dieser Prozeduren deutlich zu machen, die vom Programm unabhängig sind. Diese Teile sollten nur sehr vorsichtig und mit etwas mehr Hintergrundwissen, wie ich es in diesem Buche vermitteln kann, abgeändert werden.

7.4.2 Das Auffrischen von Fenstern

Immer wenn ein Fenster neu auf dem Bildschirm erscheint oder Teile eines Fensters, die von anderen Fenstern verdeckt waren, freigelegt werden, muß sein Inhalt aufgefrischt werden. **GetNextEvent** entdeckt solche Fenster durch Befragung des WindowManagers und erzeugt einen entsprechenden

updateEvt. Das Anwendungsprogramm ist nun dafür verantwortlich, die zunächst weißen Flächen wieder zu füllen.

Die einfachste Lösung hierfür wäre es wohl, einfach die gesamte Grafik (bzw. einen Text), die (der) im Fenster dargestellt wird, durch die entsprechenden QuickDraw-Befehle neu zu zeichnen. Bevor man damit aber beginnt, sollte man zunächst berücksichtigen, daß QuickDraw-Befehle immer im aktuellen GrafPort arbeiten und die Wirkung ein und desselben Befehls total unterschiedlich auf dem Bildschirm aussehen kann, je nachdem, welcher GrafPort gerade aktuell ist. Jedes Fenster ist aber zugleich auch ein GrafPort und die in ihm erscheinende Grafik ist stark vom Zustand dieses GrafPorts abhängig. Deshalb sollte man, bevor Zeichenoperationen ausgeführt werden, die ein bestimmtes Fenster betreffen, es vorher zum aktuellen GrafPort machen.

Wenn ein Fenster upgedatet werden muß, wird es jedoch nicht unbedingt das oberste sein. Das bedeutet, daß eventuell unregelmäßig geformte Teile des Fensters von anderen Fenstern verdeckt sind und nicht durch die Zeichen - operationen, die den Fensterinhalt auffrischen sollen, überschrieben werden dürfen. Die **clipRgn** des Fenster-GrafPorts muß deshalb so gesetzt werden, daß sie nur das sichtbare Fenster-Innere enthält.

Und schließlich muß am Ende aller Zeichenvorgänge, nachdem der korrekte Fensterinhalt wiederhergestellt wurde, dem WindowManager mitgeteilt werden, daß dieses Fenster jetzt O.K. ist. Ansonsten würde man ja bei nächster Gelegenheit wieder einen **updateEvt** dafür bekommen.

```
FUNCTION  ZeichneBild(bild: PictHandle;
                      wo: Point; scale: INTEGER);
VAR
   r:  Rect;
BEGIN
   r := bild^^.picFrame;
   OffsetRect(r,r.left,r.top);
   WITH r DO
       BEGIN
       OffsetRect(r,-left,-top);
           {   linke, obere Ecke von r
               ist jetzt gleich (0,0) }
       bottom := (bottom * scale) DIV 8;
       right  := (right  * scale) DIV 8;
       OffsetRect(r,wo.h,wo.v-bottom);
           {   linke, untere Ecke von r
               ist jetzt gleich wo }
   END;
   DrawPicture(bild,r);
END;{ZeichneBild}

PROCEDURE ZeichneInhalt(fenster: WindowPtr);
VAR
   bei:    Point;
BEGIN
   FillRect(fenster^.portRect,white);
           { löscht den ungültigen Bereich }

   bei := Point(GetWRefCon(fenster));
   ZeichneBild(wPict,bei,scale);
           { frischt das gespeicherte Bild auf }

END;{ZeichneInhalt}
```

```
PROCEDURE WindowUpdate(fenster: WindowPtr);
VAR
   savePort:   GrafPtr;
BEGIN
   GetPort(savePort); { akt. GrafPort sichern   }
   SetPort(fenster);  { Fenster aktuell  machen }
   BeginUpdate(fenster);
      IF (GetWKind(fenster) >= grafKind) THEN
         ZeichneInhalt(fenster);
   EndUpdate(fenster);
   SetPort(savePort);
                { alten GrafPort wiederherstellen }
END;{WindowUpdate}
```

WindowUpdate sichert zunächst den aktuellen GrafPort, um ihn am Ende wiederherzustellen. Das erlaubt uns, Fenster aufzufrischen, ohne den Zustand des Gesamtsystems (insbesondere von QuickDraw) zu ändern. Die Prozedur vermeidet, soweit als möglich, die sogenannten Seiten-Effekte, die ein Programm unsicher und unübersichtlich machen können. Danach wird das aufzufrischende Fenster zum aktuellen Grafport gemacht und mit **BeginUpdate** dem WindowManager die Aufgabe übertragen, die **clipRgn** korrekt zu setzen, damit die nun folgenden Zeichenoperationen kein anderes Fenster beeinträchtigen können. **EndUpdate** stellt die alte **clipRgn** wieder her und markiert das Fenster zusätzlich als aufgefrischt.

Das eigentliche Zeichnen des Fensters muß zwischen **BeginUpdate** und **EndUpdate** erfolgen. Dies leistet die Prozedur **ZeichneInhalt**, die das in der globalen Variablen **wPict** gespeicherte Bild im Fenster zeichnet. Sie zeichnet es mit der aktuellen Skalierung, die in der globalen Variablen **scale** festgehalten ist, und an dem Punkt, der im Feld **refCon** des **WindowRecords** verborgen ist. Den Wert dieses Feldes erhalten wir durch den Aufruf der Funktion **GetWRefCon**, die im WindowManager definiert ist. **GetWRefCon** liefert allerdings einen Wert vom Typ LONGINT zurück, den wir wieder unter Umgehung des Typschutzes in einen Punkt umwandeln. (Punkte und LONGINTs benötigen beide 32 Bit im Speicher, deshalb ist diese Umwandlung gefahrlos möglich.)

Im Moment verwaltet unser Programm zwar nur eine Art von Fenstern. Um aber für spätere Erweiterungen bereits vorbereitet zu sein, prüfen wir vor dem Zeichnen des Fensters erst den Fenster-Typ (im Feld **windowKind** des WindowRecords) ab. Die Funktion **GetWKind** liefert diesen Wert für einen **WindowPtr**, den man ihr als Parameter übergibt, zurück. Sie ist weiter unten beschrieben. Diesen Fenstertyp haben wir ja für die beiden vom

Programm erzeugten Fenster in der Prozedur **InitProg** vergeben. Nur für diese Fenster kennen wir den Inhalt und können ihn bei Bedarf wiederherstellen.

```
FUNCTION  GetWKind(fenster: WindowPtr): INTEGER;
BEGIN
   GetWKind := WindowPeek(fenster)^.windowKind;
END;{GetWKind}
```

7.4.3 Aktivieren und Deaktivieren von Fenstern

Das vorderste Fenster am Bildschirm – das sog. *aktive* Fenster – enthält normalerweise dasjenige Objekt, das der Benutzer gerade bearbeitet. Klickt er mit der Maus auf ein anderes Fenster, so kommt dieses nach vorn (wird *aktiviert*), und das gerade noch aktive wird "nach hinten" gelegt (wird *deaktiviert.*). Der WindowManager signalisiert diese verschiedenen Zustände von Fenstern dem Benutzer durch ein leicht geändertes Aussehen (wie im Kapitel *Fenster-Verwaltung* beschrieben). Immer wenn ein Wechsel des aktiven Fensters stattfindet, bedeutet dies auch einen Wechsel des gerade bearbeiteten Objektes. Falls dies Konsequenzen für das Verhalten des Programms hat, hat es Gelegenheit, auf einen Fensterwechsel zu reagieren, indem es **activateEvts** empfängt und bearbeitet. Eine relativ häufige Folge eines Fensterwechsels ist z.B. das Aktivieren und Deaktivieren (grau zeichnen) von Menü-Punkten, um dem Benutzer zu signalisieren, daß bestimmte Befehle nun möglich sind oder nicht möglich sind, solange dieses Fenster oben liegt.

Mit solchen Menü-Punkten sollte man aber sparsam umgehen. Sie spiegeln nämlich einen Zustand wider! Denn bei allen Design-Entscheidungen für das Programm sollte immer im Vordergrund stehen: Keine Zustände (Don't Mode Me In).

Obwohl unser noch sehr simples Beispiel-Programm eigentlich noch nicht auf einen Fenster-Wechsel reagieren müßte, habe ich zur Demonstration der dabei typischerweise anfallenden Aufgaben ein paar Menü-Änderungen in die Prozedur **WindowActivate** eingebaut.

```
FUNCTION  Sichtbar(fenster: WindowPtr): BOOLEAN;
BEGIN
   Sichtbar:= WindowPeek(fenster)^.visible;
END;{Sichtbar}
```

```
PROCEDURE WindowActivate( fenster: WindowPtr;
                            aktiv: BOOLEAN);
BEGIN
   IF aktiv THEN
        SetPort(fenster);
   IF (GetWKind (fenster) >= grafKind)
   AND aktiv THEN
        BEGIN
        IF (fenster = fenster1) THEN
             BEGIN
             DisableItem(menu[mAblage],mOeffne1);
             DisableItem(menu[mBearbeiten],mZeige1);
             IF Sichtbar(fenster2) THEN
                  EnableItem(menu[mBearbeiten],
                                 mZeige2);
             END
        ELSE IF (fenster = fenster2) THEN
             BEGIN
             DisableItem(menu[mAblage],mOeffne2);
             DisableItem(menu[mBearbeiten],mZeige2);
             IF Sichtbar(fenster1) THEN
                  EnableItem(menu[mBearbeiten],
                                 mZeige1);
             END;
        END;
END;{WindowUpdate}
```

Wie wir sehen, reagiert **WindowActivate** nur auf das Aktivieren eines Fensters. Ist der Parameter **aktiv** TRUE, so wird das Fenster mit **SetPort** zum aktuellen GrafPort gemacht und, falls es sich durch den Inhalt des Feldes **windowKind** als eins von unseren beiden Fenstern herausstellt, werden einige Menü-Punkte modifiziert.

Die Logik des Aktivierens und Deaktivierens der verschiedenen Menü-Punkte, die in **WindowActivate** dann realisiert wird, sieht ungefähr aus wie folgt: Wenn ein Fenster nach oben kommt (aktiv wird), muß es offen sein und kann demnach nicht mehr geöffnet werden. Genausowenig kann es nach oben geholt werden, da es ja schon oben liegt. Die beiden entsprechenden Menü-Befehle werden deshalb mittels **DisableItem** deaktiviert. Ist aber das andere Fenster sichtbar (geprüft mit der Prozedur **Sichtbar**), kann dieses jetzt nach oben geholt werden. Der entsprechende Menü-Punkt wird deshalb mittels **EnableItem** aktiviert.

Die hier so simple Funktion **Sichtbar** habe ich deshalb als eigene Funktion definiert, um einerseits die Lesbarkeit des Programms zu erhöhen und andererseits, weil sie nicht in jeder Programmiersprache so simpel ausfällt, wie in LisaPascal oder C. Benutzer von MacAdvantage UCSD-Pascal werden diese Funktion in wesentlich umständlicherer From neu schreiben müssen. (Es gibt hierfür mehrere Möglichkeiten, deren Realisierung ich Benutzern von MacAdvantage als Übungsaufgabe empfehle.)

7.4.4 Bearbeitung von Maus-Klicks

Das komplexeste Ereignis, das ein Programm empfangen kann, ist ein Maus-Klick. Maus-Klicks können sehr viele verschiedene Bedeutungen haben, je nachdem, wo sie passieren. Glücklicherweise ist ein Großteil der Bearbeitung eines Maus-Klicks standardisierbar und braucht nur einmal geschrieben und debugt zu werden. Der einzige Fall eines Maus-Klicks, der abhängig vom konkreten Programm ist, liegt normalerweise dann vor, wenn sich der Maus-Cursor zum Zeitpunkt des Klicks im Innern eines Fensters befand. Diesen Fall reichen wir – wie üblich – an eine andere Prozedur weiter, die die wirkliche Arbeit leistet. Der Rest der Prozedur **MausKlick** kann von nahezu jedem Programm verwendet werden.

```
PROCEDURE DoCommand(menuID,punkt: INTEGER);
BEGIN
    { wird weiter unten erläutert }
END; { DoCommand }


PROCEDURE KlickInFenster(fenster: WindowPtr;
                         wo:      Point);
BEGIN
    { wird weiter unten erläutert }
END; { KlickInFenster }


PROCEDURE SchliesseFenster(fenster: WindowPtr);
BEGIN
    { wird weiter unten erläutert }
END; { SchliesseFenster }
```

```
PROCEDURE Mausklick(wo: Point);
VAR
   gebiet:      INTEGER;
   fenster:     WindowPtr;
   menuErg:     LONGINT;
BEGIN
   gebiet := FindWindow(wo,fenster);
   CASE gebiet OF
       inMenuBar:
           BEGIN
           menuErg := MenuSelect(wo);
           DoCommand(HiWord(menuErg),
                     LoWord(menuErg));
           END; {inMenuBar}

       inSysWindow:
           BEGIN
           { Noch nicht implementiert ! }
           END; {inSysWindow}

       inDesk:
           BEGIN
           { ist uninteressant }
           END; {inDesk}

       inContent:
           BEGIN
           IF (fenster <> FrontWindow) THEN
               SelectWindow(fenster)
           ELSE
               BEGIN
               GlobalToLocal(wo);
               KlickInFenster(fenster,wo);
               END;
           END; {inContent}

       inDrag:
           BEGIN
           DragWindow(fenster,wo,dragRect);
           END; {inDrag}
```

```
        inGrow:
            BEGIN
            { Noch nicht implementiert ! }
            END; {inGrow}

        inGoAway:
            BEGIN
            IF TrackGoAway(fenster,wo) THEN
                SchliesseFenster(fenster);
            END; {inGoAway}
        END;{CASE}
END;{Mausklick}
```

MausKlick besteht im wesentlichen aus einer großen CASE-Anweisung, die zwischen den verschiedenen Gebieten unterscheidet, die der Maus-Klick getroffen haben könnte. Die Einteilung des Bildschirms in Gebiete brauchen wir dabei nicht einmal selbst vorzunehmen; das erledigt die Funktion **FindWindow** des WindowManagers. **FindWindow** liefert zu einem Punkt in globalen Koordinaten einen Code zurück, der das getroffene Gebiet kennzeichnet und im VAR-Parameter **fenster** das getroffene Fenster – falls der Maus-Klick auf ein Fenster traf. Die folgende Abbildung listet noch einmal all die verschiedenen Bereiche auf, die ein Maus-Klick treffen kann. Für die genaue Abgrenzung der verschiedenen Bereiche voneinander verweise ich auf das Kapitel *Fenster-Verwaltung*.

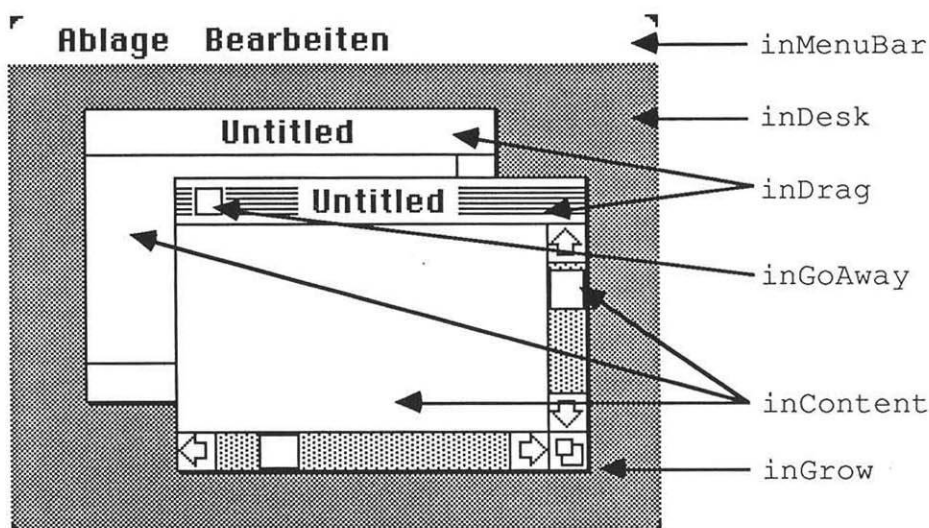


Bild 7 - 2: *Bereiche, die FindWindow unterscheidet*

Traf der Maus-Klick in die Menü-Leiste, ruft **MausKlick** die Funktion **MenuSelect** auf, die als Ergebnis das Menü und den Menü-Punkt, den der Benutzer ausgewählt hat, zurückgibt. Diesen gepackten 32-Bit-Wert zerlegen wir in der bereits bekannten Weise in zwei 16-Bit-Zahlen mit denen wir dann die Prozedur **DoCommand** aufrufen. Das alles läuft recht ähnlich wie die Bearbeitung eines **keyDown**-Events in der Prozedur **BearbeiteErg** ab.

Traf der Maus-Klick in das Innere eines Fensters (**inContent**), prüfen wir zunächst, ob das bewußte Fenster das oberste ist. Falls das nicht der Fall ist, holen wir es mit **SelectWindow** nach oben und tun sonst nichts. Lag das Fenster bereits oben, konvertieren wir die Koordinaten des Maus-Klicks in das lokale Koordinatensystem des Fensters und reichen den Maus-Klick an die Prozedur **KlickInFenster** weiter.

Bei einem Maus-Klick auf die Titel-Leiste eines Fensters (**inDrag**) reichen wir die Koordinaten des Klicks einfach an **DragWindow** weiter. Diese Prozedur des WindowManagers leistet alles, was zur Verschiebung von Fenstern nötig ist. Sie benötigt als zweiten Parameter ein Rechteck, das in globalen Koordinaten Grenzen angibt, innerhalb derer das Fenster verschoben werden darf. Wir übergeben hierfür das Rechteck **dragRect**, das

in **InitMac** initialisiert wurde und einen Bereich umschließt, der etwas kleiner als der gesamte Bildschirm ist.

Einen Klick in das Schließ-Kästchen eines Fensters reichen wir an die Prozedur **SchliesseFenster** weiter, die programmabhängig entscheidet, was zu tun ist, um dieses Fenster vom Bildschirm zu entfernen.

Drei Bereiche, in die Maus-Klicks fallen können, ignorieren wir (zunächst noch) völlig: **inSysWindow, inGrow** und **inDesk. InSysWindow** signalisiert einen Klick in das Fenster eines Schreibtisch-Zubehörs (*Desk Accessories,* im folgenden meist kurz "DAs" genannt) und **inDesk** einen Klick in den Bildschirmhintergrund. Ein Klick in den (meist grauen) Hintergrund wird nur von den wenigsten Programmen (meist Spielen) bearbeitet, und Fenster von DAs brauchen wir nicht zu berücksichten, solange unser Programm kein Apfel-Menü hat, durch das man sie aufrufen kann. Klicks in das Grow-Icon eines Fensters (**inGrow**) behandeln wir erst in einer späteren Ausbaustufe unseres Programms.

Man beachte bei den verschiedenen Regionen eines Fensters, die von einem Maus-Klick getroffen werden können, übrigens die unterschiedliche Behandlung des obersten Fensters (das wir mittels **FrontWindow** feststellen)! Einen Klick in die Titel-Leiste (**inDrag**) behandeln wir immer gleich – egal ob das Fenster unten oder oben liegt. Einen Klick in das Schließ-Kästchen (**inGoAway**) meldet der WindowManager nur für das oberste Fenster. Klicks ins Fenster-Innere (**inContent**) müssen wir selbst korrekt behandeln. Normalerweise haben solche Klicks nur beim obersten Fenster eine spezielle Wirkung. Hinten liegende Fenster werden durch einen Klick in den Inhalt erst nach vorne geholt.

Mit dieser Beschreibung der Prozedur **MausKlick** wäre die Beschreibung der "allgemeingültigen" Prozeduren dann erst einmal abgeschlossen. Die nun folgenden Abschnitte beschäftigen sich mit den Prozeduren, die das spezielle (sehr simple) Verhalten des Beispielprogramms realisieren.

7.5 Ein Klick im Fenster

Die Prozedur **KlickInFenster** realisiert die wichtigsten, weil auffälligsten Verhaltensweisen des Beispielprogramms. Ein Maus-Klick in das oberste Fenster führt stets dazu, daß das in **wPict** gespeicherte kleine Bild im Innern des Fensters gezeichnet wird. Und zwar liegt stets die linke untere Ecke dieses Bilds an der Stelle, an der sich der Maus-Cursor gerade befindet.

Solange die Maustaste gedrückt bleibt, folgt das Bild dem Cursor. Wird sie gelöst, bleibt das Bild an der zuletzt erreichten Stelle stehen. Diese Stelle wird dann im Feld **refCon** des entsprechenden WindowRecords für die Verwendung durch andere Prozeduren gespeichert.

```
PROCEDURE KlickInFenster(fenster: WindowPtr;
                                  wo: Point);
VAR
   punkt:       Point;
BEGIN
   IF (GetWKind(fenster) = grafKind) THEN
       BEGIN
       punkt := Point(GetWRefCon(fenster));
       ZeichneBild(wPict,punkt,scale);
       ZeichneBild(wPict,wo,scale);
       WHILE StillDown DO
           BEGIN
           ZeichneBild(wPict,wo,scale);
           GetMouse(wo);
           ZeichneBild(wPict,wo,scale);
           END;
       SetWRefCon(fenster,LONGINT(wo));
       END;
END; {KlickInFenster}
```

KlickInFenster beginnt wie üblich mit der Feststellung des Fenstertyps. Handelt es sich um eins "unserer" beiden Fenster, wird danach mit dem ersten **ZeichneBild**-Aufruf das Bild an der letzten Position gelöscht. (Dieses Löschen geschieht unter Ausnutzung des QuickDraw-Zeichenmodus **patXor**, in dem das Bild gezeichnet wird. Das erste Zeichnen des Bildes an einer bestimmten Stelle läßt es erscheinen, das nächste Zeichnen an derselben Stelle läßt es wieder verschwinden.) Der zweite **ZeichneBild**-Aufruf zeichnet das Bild an der Stelle des Maus-Klicks – die im Parameter **wo** übergeben wurde – neu.

Die folgende WHILE-Schleife bleibt so lange aktiv, wie der Benutzer die Maustaste gedrückt läßt. In ihr wird stets zunächst das Bild an der letzten Position gelöscht, die neue Position des Cursors festgestellt und das Bild dann an diesem Punkt neu gezeichnet. Man achte übrigens darauf, daß der Aufruf **GetMouse** die Position des Cursors lokal (d.h. im Koordinatensystem des aktuellen GrafPorts) wiedergibt! Ein Umwandeln dieser Koordinaten mittels **GlobalToLocal**, wie es mit den Maus-Koordinaten im EventRecord nötig war, braucht hier nicht zu erfolgen. Da im

allgemeinen immer das oberste Fenster zugleich auch der aktuelle GrafPort ist, kann **KlickInFenster** nur für das oberste Fenster richtig funktionieren.

Nachdem die WHILE-Schleife verlassen wurde – die Maustaste nun also gelöst ist – werden die Koordinaten der letzten Stelle, an der das Bild gezeichnet wurde, wieder in das Feld **refCon** des WindowRecords gepackt. Dies ist vor allem für das Auffrischen des Fensters nötig, aber auch für den nächsten Aufruf von **KlickInFenster**. Damit **KlickInFenster** das alte Bild löschen kann, muß es ja seine Position kennen. Und da wir mehrere (im Moment 2) Fenster verwalten, in denen das Bild gezeichnet werden kann, sollte das Programm fensterspezifische Daten auch direkt beim WindowRecord verwalten.

Der WindowManager reserviert für solche Daten aber nur einen Platz von 32 Bit. Dies reicht für einen Punkt noch aus; für mehr aber schon nicht mehr. Muß man größere Datenmengen pro Fenster speichern, so werden diese im allgemeinen in einer separaten Datenstruktur abgelegt auf die **refCon** dann mit einem Zeiger oder einem Handle verweist. Wer mit diesem Konzept experimentieren möchte, kann ja versuchen, zusammen mit dem aktuellen Punkt, an dem das Bild gezeichnet wird, die aktuelle Skalierung des Bildes in einem Record abzulegen, das dynamisch im Heap erzeugt wird. Dies hat allerdings Änderungen an mehreren Stellen der schon vorgestellten Prozeduren zur Folge, die man zunächst alle heraussuchen und markieren sollte.

7.6 Das Schließen eines Fensters

Obwohl das Schließen eines Fensters im ersten Rahmenprogramm eine recht simple Sache ist, wird es doch in eine separate Prozedur ausgelagert, da es stark programmabhängig ist, was beim Schließen eines Fensters passiert.

```
PROCEDURE SchliesseFenster(fenster: WindowPtr);
BEGIN
   IF (GetWKind(fenster) = grafKind) THEN
      BEGIN
         HideWindow(fenster);
         IF (fenster = fenster1) THEN
            BEGIN
            EnableItem(menu[mAblage],Oeffne1);
            DisableItem(menu[mBearbeiten],Zeige1);
            END
         ELSE IF (fenster = fenster2) THEN
            BEGIN
            EnableItem(menu[mAblage],Oeffne2);
            DisableItem(menu[mBearbeiten],Zeige2);
            END;
      END;
END; {SchliesseFenster}
```

SchliesseFenster beginnt wie üblich mit der Feststellung des Fenstertyps. Handelt es sich um eins "unserer" beiden Fenster, verbergen wir es mit **HideWindow**. Damit der Benutzer dieses Fenster wieder öffnen, es aber nicht nach vorne holen kann, aktivieren bzw. deaktivieren wir danach die entsprechenden Menü-Punkte.

7.7 Menü-Befehle

Die Prozedur **DoCommand** nimmt für die Reaktion auf Menü-Befehlen die gleiche zentrale Stellung ein, wie **BearbeiteEreig** sie für die Reaktion auf Ereignisse besitzt. **DoCommand** sollte nach jeder Menü-Auswahl des Benutzers durchlaufen werden, ist zugleich aber auch so geschrieben, daß sie problemlos von jeder anderen Prozedur aus aufgerufen werden kann. Wird diese Möglichkeit nicht genutzt, wird **DoCommand** in einem "ordentlichen Macintosh-Programm" nur an zwei Stellen anfgerufen: in **BearbeiteErg** nach einem Tastendruck bei gleichzeitig gedrückter Befehls-Taste und in **MausKlick** nach einem Klick in die Menü-Leiste.

Neben dem Verteilen der Menü-Befehle auf andere Prozeduren hat **DoCommand** auch noch die Aufgabe, den Titel des gerade gewählten Menüs so lange invertiert zu halten, wie die Ausführung des ausgewählten Befehls andauert, und danach den "Normalzustand" wiederherzustellen.

```
CONST   { Menü- und Punkt-Nummern der beiden Menüs }
    mAblage=          1;
        mOeffne1=     1;
        mOeffne2=     2;
        { Strich=     3; }
        mBeenden=     4;

    mBearbeiten=      2;
        mGroesser=    1;
        mKleiner=     2;
        { Strich=     3; }
        mZeige1=      4;
        mZeige2=      5;

PROCEDURE InvalFenster(fenster: WindowPtr);
VAR
    savePort:   GrafPtr;
    r:          Rect;
BEGIN
    GetPort(savePort);
    SetPort(fenster);
    r := fenster^.port.portRect;
    InvalRect(r);
    SetPort(savePort);
END;{InvalFenster}
```

```
PROCEDURE DoCommand(menuID,punkt: INTEGER);
BEGIN
   HiliteMenu(0);
   HiliteMenu(menuID);
   CASE menuID OF
      mAblage:
         BEGIN
         CASE punkt OF
            mOeffne1:
               BEGIN
               ShowWindow(fenster1);
               SelectWindow(fenster1);
               END;
            mOeffne2:
               BEGIN
               ShowWindow(fenster2);
               SelectWindow(fenster2);
               END;
            mBeenden:
               ende := TRUE;
            END;{CASE punkt}
         END;{mAblage}

      mBearbeiten:
         BEGIN
         CASE punkt OF
            mGroesser:
               BEGIN
               scale := scale * 2;
               EnableItem(menu[mBearbeiten],
                          mKleiner);
               IF (scale >= 64) THEN
                  BEGIN
                  scale := 64;
                  DisableItem(
                          menu[mBearbeiten],
                          mGroesser);
                  END;
               InvalFenster(fenster1);
               InvalFenster(fenster2);
               END;
```

```
                    mKleiner:
                        BEGIN
                        scale := scale DIV 2;
                        EnableItem(menu[mBearbeiten],
                                   mGroesser);
                        IF (scale <= 1) THEN
                            BEGIN
                            scale := 1;
                            DisableItem(
                              menu[mBearbeiten],
                              mKleiner);
                            END;
                        InvalFenster(fenster1);
                        InvalFenster(fenster2);
                        END;
                    mZeige1:
                        SelectWindow(fenster1);
                    mZeige2:
                        SelectWindow(fenster2);
                    END;{CASE punkt}
                END;{mBearbeiten}
            END; {CASE menuID}
    HiliteMenu(0);
END; {DoCommand}
```

DoCommand besteht aus einer geschachtelten CASE-Anweisung. Die äußere CASE-Anweisung unterscheidet nach Menü-Nummern, und für jede Menü-Nummer existiert wieder eine CASE-Anweisung, die nach den einzelnen Punkten unterscheidet. Einfache Menü-Befehle werden direkt in den Punkten der inneren CASE-Anweisung realisiert, komplexe werden an andere Prozeduren weitergereicht. Wir haben es in unserem Beispiel nur mit relativ einfachen Befehlen zu tun, die wir alle direkt in **DoCommand** implementieren.

Bevor wir uns allerdings um die Befehle selbst kümmern, heben wir den Titel des Menüs, in dem sich der Befehl befindet, mittels **HiliteMenu** hervor. Am Ende der Bearbeitung stellen wir das normale Aussehen der Menü-Leiste mittels **HiliteMenu(0)** wieder her. Dieser Teil von **DoCommand** ist allgemeingültig, der Rest programmspezifisch. Der doppelte **HiliteMenu**-Aufruf, wie er am Anfang dieser Implementierung von **DoCommand** auftaucht, ist nur dann nötig, wenn **DoCommand** auch von anderen Prozeduren als **BearbeiteErg** und **MausKlick** aufgerufen werden kann. In

Programmen, in denen dies nicht vorkommt, können beide **HiliteMenu**-Aufrufe fortfallen. Ist **DoCommand** nämlich die Folge einer Menü-Auswahl des Benutzers — egal ob mit der Maus oder der Tastatur — hebt der MenuManager von sich aus bereits den entsprechenden Menü-Titel hervor.

Nun zur Ausführung der Menü-Befehle: Auf die ***Öffne...***-Befehle im ersten Menü reagieren wir einfach, indem wir das entsprechende Fenster mittels **ShowWindow** sichtbar machen. Der Befehl ***Beenden*** bewirkt, daß die globale Variable **ende** auf TRUE gesetzt wird und die Event-Schleife im Hauptprogramm verlassen wird, sobald das Programm das nächste Mal auf "**UNTIL ende**" stößt.

Die Ausführung der Befehle ***Größer*** und ***Kleiner*** ist schon etwas schwieriger. Zunächst wird einfach der Skalierungs-Faktor **scale** verdoppelt bzw. halbiert. Stößt er dabei an die untere bzw. obere Grenze, so wird der entsprechende Menü-Befehl deaktiviert, um dem Benutzer zu zeigen, daß er das Bild nicht noch größer bzw. kleiner machen kann. Der "entgegengesetzte" Menü-Befehl wird aber auf alle Fälle aktiviert, da es ja z.B. immer möglich ist, die Zeichnung zu vergrößern, wenn sie gerade verkleinert wurde. Nach einer Änderung des Skalierungs-Faktors entspricht der aktuelle Inhalt beider Fenster nicht mehr der "Wirklichkeit". Deshalb werden nach jeder Änderung von **scale** beide Fenster vollständig invalidiert.

Die Befehle ***1 nach vorn*** und ***2 nach vorn*** sind wieder etwas leichter zu befolgen. Das entsprechende Fenster wird einfach mittels **SelectWindow** zum obersten Fenster gemacht.

7.8 Das komplette Programm

Statt eines kompletten Listings — das recht viel Platz benötigen würde — folgt nun eine Roh-Skizze des ersten Beispiel-Programms. Ich liste alle Konstanten, Variablen und Prozedur-Köpfe in der Reihenfolge, in der sie im Programm auftauchen müßten, damit der Pascal-Compiler es akzeptiert. Benutzer von C und Modula-2 haben es da einfacher, da Compiler für diese Sprachen meist keine Reihenfolge-Regeln in der Form von Pascal haben. Trotzdem macht es Programme auch in diesen Sprachen übersichtlicher, wenn Prozeduren definiert werden, bevor sie benutzt werden. (Zum Verständnis eines noch unbekannten Programms ist es jedoch oft besser, umgekehrt vorzugehen — also vom Hauptprogramm zu den Prozeduren — weshalb ich dieses Kapitel auch so strukturiert habe.)

```
CONST
    mAblage      = 1;
        mOeffne1= 1;
        mOeffne2= 2;
        mBeenden= 4;

    mBearbeiten = 2;
        mGroesser=1;
        mKleiner= 2;
        mZeige1=  4;
        mZeige2=  5;

    lastMenu     = 2;

    grafKind     = 8;


VAR
    ende:          BOOLEAN;
    menu:          ARRAY [1..lastMenu] OF menuHandle;

    dragRect:      Rect;

    fenster1:      WindowPtr;
    fenster2:      WindowPtr;

    wPict:         PicHandle;
    scale:         INTEGER;

    fMouse:        BOOLEAN;
    fShift:        BOOLEAN;
    fShLock:       BOOLEAN;
    fOption:       BOOLEAN;
    fCommand:      BOOLEAN;
    fActive:       BOOLEAN;

    dasEreignis: EventRecord;

{   Es folgen nun zunächst die Hilfsprozeduren, die
 hauptsächlich das Arbeiten mit Fenstern etwas
    einfacher machen
}
FUNCTION   Sichtbar(fenster: WindowPtr): BOOLEAN;
```

```
PROCEDURE  SetWKind(fenster: WindowPtr;
                    kind: INTEGER);
FUNCTION   GetWKind(fenster: WindowPtr): INTEGER;
PROCEDURE  InvalFenster(fenster: WindowPtr);


{   Es folgen die stark Programm-abhängigen
    Prozeduren und Funktionen, die das Bild im
    Fensterinnern zeichnen und bewegen.
}
FUNCTION   CreatePict: PicHandle;
PROCEDURE  ZeichneBild(bild: PicHandle;
                   wo: Point; scale: INTEGER);
FUNCTION   KlickInFenster(fenster: WindowPtr;
                          wo: Point);


{   Die folgende Prozedur ist zwar Programm-
    abhängig, enthält aber viele übertragbare Teile.
}
PROCEDURE  ZeichneInhalt(fenster: WindowPtr);


{   Die folgenden Prozeduren erledigen zusammen
    den größten Teil der Analyse und Abarbeitung der
    vom Benutzer kommenden Ereignisse. Sie sind in
    hohem Maße übertragbar auf jedes "echte
    Macintosh-Programm".
}
PROCEDURE  WindowUpdate(fenster: WindowPtr);
PROCEDURE  WindowActivate(fenster: WindowPtr;
                          aktiv: BOOLEAN);
PROCEDURE  SchliesseFenster(fenster: WindowPtr);
PROCEDURE  DoCommand(menuID,punkt: INTEGER);
PROCEDURE  MausKlick(wo: Point);
PROCEDURE  BearbeiteEreig;


{   Die letzten beiden Prozeduren dienen der
    Initialisierung der Programmumgebung. InitMac
    ist hochgradig übertragbar InitProg sehr
    programmspezifisch.
}
```

```
PROCEDURE  InitMac;
PROCEDURE  InitProg;
```

7.9 Schlußbemerkung

Das in diesem Kapitel vorgestellte Programm ist ein erster Einstieg in das Erstellen echter MacProgramme. Es leistet zwar noch nicht sehr viel, aber das Wichtigste — die Haupt-Event-Schleife — läuft. Es handelt sich im Moment allerdings noch um ein "Rahmenprogramm der Stufe 0", da wichtige Aspekte des Standardverhaltens eines Macintosh-Programms noch nicht unterstützt werden.

Was aber wichtig ist, ist die Struktur des Programms. Diese ist jetzt schon korrekt und sieht im wesentlichen bereits so aus wie auch das letzte Beispielprogramm in diesem Buch, in dem dann alle Aspekte des Standardverhaltens eines Macintosh-Programms unterstützt werden. Viele Stellen, die bis jetzt leer geblieben sind, müssen nur (!) noch gefüllt werden.

Die wichtigsten Dinge, die unser Programm noch nicht kann, sind:

- das Ändern der Größe von Fenstern
- das Ablegen aller landesabhängigen Texte etc. in Resources
- die Unterstützung von Schreibtisch-Utensilien.

Während nicht jedes (aber fast jedes) Programm Fenster benötigt, die auch vergrößert und verkleinert werden können, ist die Unterstützung von Schreibtisch-Utensilien einfach ein echtes Muß. Es gibt inzwischen so viele dieser wirklich nützlichen kleinen Werkzeuge, daß es für den Benutzer einfach lästig wäre, in einem Programm darauf verzichten zu müssen.

Das Ablegen von Texten für Menüs, Fenstertitel etc. in Resourcen — also getrennt vom Programmtext — ist auch keine Sache, die unbedingt nötig wäre. Aber ganz davon abgesehen, daß es praktisch dem "Guten Ton" unter MacProgrammierern entspricht, dies so zu machen, erleichtert es uns auch das Programm selbst. Und sollte ein Programm wirklich einmal international vertrieben werden, kann es problemlos an die Landessprache angepaßt werden.

Das Beheben dieser drei noch verbleibenden Mängel der ersten Fassung des Rahmenprogramms wird die Aufgabe der restlichen drei Kapitel des zweiten

Teils dieses Buches sein!

8 Die erste Ausbaustufe

Nachdem im letzten Kapitel das Rahmenprogramm in seiner simpelsten Form vorgestellt wurde, soll es in diesem und den folgenden Kapiteln des zweiten Buchteils weiter ausgebaut werden. Die folgenden Abschnitte dieses Kapitels beschreiben die erste Ausbaustufe, die sich hauptsächlich mit dem Vergrößern und Verkleinern von Fenstern beschäftigen wird.

8.1 Was soll das Programm leisten?

Im letzten Kapitel wurde die Prozedur **MausKlick** vorgestellt. Sie wurde immer dann von der Haupt-Event-Schleife aus aufgerufen, wenn über die Prozedur **GetNextEvent** ein **mouseDown**-Event empfangen wurde. In MausKlick wird überprüft (mittels **FindWindow**), welches Gebiet des Bildschirms von diesem Klick getroffen wurde und abhängig davon im Programm weiterverzweigt.

Einer der Fälle, den wir in **MausKlick** noch nicht bearbeitet hatten, war **inGrow** gewesen, ein Klick in das **Grow-Icon** in der rechten unteren Ecke eines offenen Fensters. Üblicherweise bedeutet ein Klick in dieses Grow-Icon den Beginn einer Größenveränderung eines Fensters. Dabei folgt, solange die Maustaste gedrückt bleibt, eine graue Umrißlinie den Mausbewegungen, die die neue Größe des Fensters andeutet, die dieses erhalten würde, wenn in diesem Moment die Maustaste gelöst würde.

Um dem Benutzer aber überhaupt zu signalisieren, daß er das Fenster vergrößern bzw. verkleinern kann, muß zuvor auch das Grow-Icon in der unteren rechten Ecke des Fensters gezeichnet werden.

Damit dieses Grow-Icon aber nicht so allein in der Fensterecke bleibt, wird unser neues Programm auch gleich schon am unteren und am rechten Rand des Fensters die beiden Streifen andeuten, in denen bei einem Dokument-Fenster üblicherweise die Rollbalken liegen, mit denen man den Fensterinhalt verschieben (*scrollen*) kann. Im Moment werden diese beiden Streifen noch

weiß bleiben, da noch kein Scrollen des Fensters unterstützt wird. Der Inhalt dieser Streifen bleibt stets weiß, d.h. die Grafik, die im Innern des Fensters gezeichnet wird, erstreckt sich nun nicht mehr bis zum Fensterrand, sondern nur noch bis zum Rand der angedeuteten Rollbalken.

Die folgende Abbildung zeigt, wie gerade die Größe des obersten der beiden Fenster des Rahmenprogramms verändert wird. Wie deutlich zu sehen ist, sieht das Grow-Icon des hinten liegenden Fensters anders aus (bzw. ist nur durch einen Rahmen angedeutet), als das des gerade bearbeiteten Fensters.

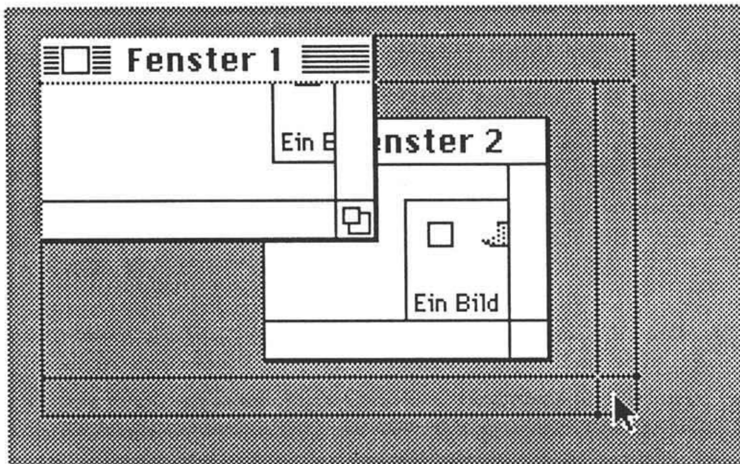


Bild 8 - 1: *Das Vergrößern eines Fensters*

8.2 Überblick über die nötigen Änderungen

Dieses im vorangehenden Abschnitt beschriebene Verhalten des Programms verlangt einige Änderungen am Programm **Rahmen0**, das im letzten Kapitel implementiert wurde. Das dadurch entstehende Programm wird originellerweise den Namen **Rahmen1** tragen.

In der Initialisierungsphase des Programms müssen wir zunächst dafür sorgen, daß das Gebiet der beiden Fenster, in dem die gespeicherte Grafik gezeichnet werden kann, verkleinert wird. Die Grafik darf ja nicht über den Bereich der angedeuteten Rollbalken gezeichnet werden. Hierfür bietet sich die **clipRgn** der GrafPorts der beiden Fenster an. Diese Möglichkeit, die

Grafikausgaben zu begrenzen, sind in **Rahmen0** überhaupt nicht verwendet worden, da sich die Grafik ja bis an den Rahmen der Fenster erstrecken durfte. Die Begrenzung auf dieses Rechteck aber erledigte bereits das Feld **portRect** der GrafPorts der beiden Fenster, außerhalb dessen ja keine Wirkungen von QuickDraw-Operationen eintreten (vgl. hierzu bei Unklarheiten das Kapitel *QuickDraw*).

Die wichtigste Änderung liegt natürlich in der Prozedur **MausKlick**, in der jetzt auch der Fall **inGrow** der CASE-Anweisung berücksichtigt wird. Da der WindowManager für die Größenänderung von Fenstern keine vergleichbare bequeme Prozedur wie **DragWindow** zum Verschieben von Fenstern anbietet, wird die Prozedur **WindowGrow** im Programm **Rahmen1** realisiert. **WindowGrow** wird alle nötigen Maßnahmen zum Ändern der Fenstergröße nach einem Maus-Klick in das Grow-Icon unternehmen, die programmunabhängig sind.

Und beim Zeichnen des Fenster-Inhalts schließlich werden wir ebenfalls kleinere Änderungen machen müssen, damit das Grow-Icon und die beiden leeren Rollbalken im Fenster erscheinen. Im wesentlichen wird dies die Prozedur **DrawGrowIcon** leisten, die zum WindowManager gehört. **Rahmen1** wird aber selbst dafür sorgen müssen, daß das Innere der leeren Rollbalken gelöscht wird.

8.3 Änderungen in der Initialisierungsphase

Die Möglichkeit der Größenänderung von Fenstern verlängert einige kleinere Modifikationen an InitMac und InitProg. Zudem benötigen wir eine neue globale Variable, die in InitMac initialisiert wird.

8.3.1 Änderungen an InitMac

In **InitMac** wird nun zusätzlich zur Initialisierung der ToolBox und der Variablen **dragRect** noch das Rechteck **growRect** initialisiert. **GrowRect** wird später im Programm benötigt, um der Funktion **GrowWindow** des **WindowManagers** (nicht zu verwechseln mit **WindowGrow** aus **Rahmen1**) mitzuteilen, innerhalb welcher Grenzen sich die horizontalen und vertikalen Abmessungen eines Fensters bewegen können.

```
VAR     dragRect,
        growRect:    Rect

PROCEDURE InitMac;
BEGIN
    InitGraf(@thePort);
    InitFonts;
    InitWindows;
    FlushEvents(everyEvent,0);
    TEInit;
    InitDialogs(NIL);
    dragRect := screenBits.bounds;
    dragRect.top := dragRect.top + 20;
            { Menü-Leiste oben abziehen }
    InsetRect(dragRect,4,4);
            { ringsherum 4 Pixel kleiner}

    growRect := dragRect;
    growRect.left := 100;
    growRect.top  := 50;
            { right und bottom können bleiben }

    InitCursor;
END;{InitMac}
```

GrowRect wird zunächst gleich **dragRect** gesetzt, ist also auch abhängig von der Größe des Bildschirms (über **screenBits.bounds**). **InitMac** läßt **growRect.bottom** und **growRect.right** (Maxima für die vertikale bzw. horizontale Fenstergröße) unverändert und setzt **left** auf 100 (Minimum für die Breite eines Fensters) und **top** auf 50 (Minimum für die Höhe eines Fensters).

Diese Grenzen für Fenstergrößen sind nicht ganz unabhängig von einem speziellen Programm, sind aber recht plausible Werte, die für die meisten Programme genau richtig sein dürften. Fenster können damit fast so groß wie der gesamte Bildschirm werden, aber nie kleiner als 100 x 50 Punkte. (Immer vorausgesetzt allerdings, der Bildschirm ist nicht kleiner als 100 x 50, wovon man aber wohl auch für zukünftige Mac-Nachfolger von Apple getrost ausgehen kann.)

8.3.2 Änderungen an InitProg

InitProg muß – zusätzlich zu seinen alten Aufgaben – nur dafür sorgen, daß die **clipRgn** der GrafPorts unserer beiden Fenster so gesetzt wird, daß sämtliche QuickDraw-Operationen, die das Fenster-Innere zeichnen, die beiden Rollbalken und das Grow-Icon unbeeinträchtigt lassen.

Dazu brauchen vor dem Ende von **InitProg** nur die beiden folgenden Prozedur-Aufrufe eingefügt werden:

```
    ......
    FensterAendern(fenster1);
    FensterAendern(fenster2);

END; { InitProg }
```

FensterAendern wird im Abschnitt 8.4.3 näher erläutert und führt alle nötigen Operationen zur Änderung der **clipRgn** des entsprechenden Fensters durch. **FensterAendern** wird nach jeder Änderung der Fenstergröße aufgerufen und wird deshalb zusammen mit den Prozeduren, die dafür zuständig sind, beschrieben. Die Prozedur ändert die **clipRgn** dann in Abhängigkeit von der aktuellen Fenstergröße.

8.4 Ein Klick in das Grow-Icon

Die wichtigsten Änderungen an **Rahmen0** haben natürlich mit dem Maus-Klick in das **GrowIcon** zu tun. Einen solcher Maus-Klick ist bis jetzt ingnoriert worden, führt nun aber zu zwei Prozedur-Aufrufen, die dann dafür sorgen, daß der Benutzer die Fenstergröße ändern kann.

8.4.1 Änderungen an MausKlick

Die nötigen Änderungen an MausKlick bestehen nur aus zwei Prozedur-Aufrufen. Die eigentliche Arbeit wird dann in diesen Prozeduren geleistet.

```
PROCEDURE Mausklick(wo: Point);
VAR
   gebiet:       INTEGER;
   fenster:      WindowPtr;
   menuErg:      LONGINT;
BEGIN
   gebiet := FindWindow(wo,fenster);
   CASE gebiet OF
       .........
       { Der Anfang bleibt unverändert }
       inGrow:
           BEGIN
               IF fenster <> FrontWindow THEN
                   SelectWindow(fenster)
               ELSE
                   BEGIN
                   WindowGrow(fenster,wo,growRect);
                   AenderFenster(fenster);
                   END;
           END;  {inGrow}

       { Der Rest bleibt unverändert }
       .........
       END;{CASE}
END;{Mausklick}
```

Ein Maus-Klick im Innern eines hinten liegenden Fensters bewirkt nur, daß dieses nach vorne kommt und sonst nichts — auch, wenn der Klick ins Grow-Icon traf. **MausKlick** fragt deshalb zunächst ab, ob es sich bei dem getroffenen Fenster um das vorderste handelt, und holt es mit **SelectWindow** nach vorn, wenn dies nicht der Fall ist. Nur wenn das Fenster schon an oberster Stelle lag, werden die beiden Prozeduren WindowGrow und AenderFenster aufgerufen. **WindowGrow** ist für die programmunabhängige Arbeit der eigentlichen Größen-Änderung zuständig, und **AenderFenster** führt die danach nötigen programmabhängigen Änderungen durch.

8.4.2 Die Prozedur WindowGrow

WindowGrow entspricht in etwa der Prozedur **DragWindow**. Genau wie **DragWindow** nach einem Klick in die Titel-Leiste eines Fensters alles Nötige übernimmt, um das Fenster zu verschieben, ist **WindowGrow** in

entsprechender Weise für Maus-Klicks ins Grow-Icon eines Fensters zuständig. Nur ist **WindowGrow** nicht in der ToolBox, sondern muß von jedem Programm selbst realisiert werden.

```
PROCEDURE InvalGrow(fenster:  WindowPtr);
BEGIN
   { wird weiter unten erläutert }
END; {InvalGrow}


PROCEDURE WindowGrow(fenster:  WindowPtr;
                     mausPt:  Point;
                     grenzen: Rect);
VAR
   growErg:    LONGINT;
BEGIN
   InvalGrow(fenster);
   growErg := GrowWindow(fenster,mausPt,grenzen);
   SizeWindow(fenster,
              LoWord(growErg),
              HiWord(growErg),
              TRUE);
   InvalGrow(fenster);
END;{WindowGrow}
```

WindowGrow besteht im wesentlichen aus zwei ToolBox-Aufrufen: **GrowWindow** und **SizeWindow**. **GrowWindow** ändert nicht wirklich die Größe eines Fensters, sondern folgt nur (innerhalb des von **grenzen** gesetzten Rahmens), solange die Maus-Taste gedrückt ist, den Mausbewegungen. Währenddessen zeigt sie kontinuierlich dem Benutzer durch eine graue Umrißlinie an, welche neue Größe das Fenster haben würde, wenn er jetzt die Maus-Taste losließe. Sobald die Maustaste gelöst wird, liefert **GrowWindow** als Funktionsergebnis die neue, vom Benutzer gewünschte Fenstergröße an die aufrufende Prozedur zurück. Die neue Breite des Fensters liegt in den unteren und die neue Höhe in den oberen 16 Bits des 32 Bits breiten Funktionsergebnisses.

Nachdem dieser 32-Bit-Wert mit den bereits bekannten Hilfsroutinen **LoWord** und **HiWord** zerlegt wurde, wird dann die Größe des Fensters mit der ToolBox-Prozedur **SizeWindow** wirklich geändert (**GrowWindow** ändert trotz seines Namens das Fenster überhaupt nicht). Indem als dritter Parameter TRUE an **SizeWindow** übergeben wird, teilt man dem **WindowManager** mit, daß er den Bereich, um den das Fenster eventuell

größer geworden ist, als ungültig markieren soll. **GetNextEvent** wird dann bei nächster Gelegenheit einen **UpdateEvent** dafür liefern.

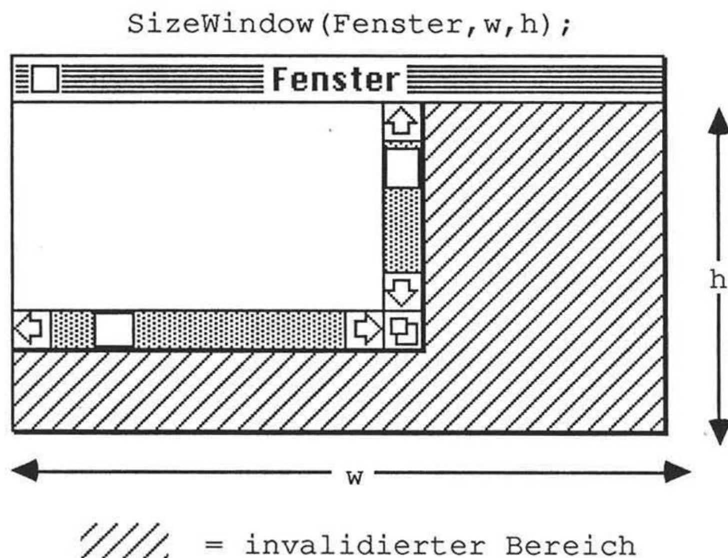


Bild 8 - 2: *Wirkung von SizeWindow*

Vor und nach dem Verändern der Fenstergröße rufen wir die Prozedur **InvalGrow** auf. Sie ist programmabhängig und dafür gedacht, diejenigen Gebiete des Fensters als ungültig zu markieren, die von einer Änderung der Fenstergröße betroffen sind – in Rahmen1 sind das natürlich die beiden leeren Rollbalken und das Grow-Icon; in anderen Programmen könnten es aber auch noch andere Gebiete sein.

Diese Gebiete, die **InvalGrow** invalidiert, sehen vor und nach einer Änderung der Fenstergröße anders aus. Sie werden aber teilweise nicht – je nachdem, ob das Fenster größer oder kleiner geworden ist – von **SizeWindow** als ungültig markiert. Die folgende Abbildung zeigt dies schematisch am Beispiel der beiden Rollbalken.

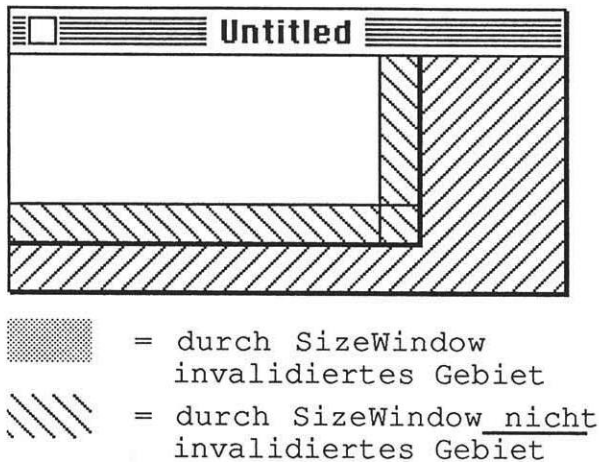


Bild 8 - 3: *Durch eine Größenänderung betroffene Gebiete*

Wie Bild 8 – 3 deutlich zeigt, werden durch SizeWindow zwar die Gebiete als ungültig markiert, die vor einer Vergrößerung außerhalb des Fenster-Innern lagen, jedoch keine Gebiete, die vor einer beliebigen Änderung der Fenstergröße innerhalb des Fenster-Innern lagen. Enthält ein Fenster aber z.B. Rollbalken, so müssen deren Flächen ebenfalls invalidiert werden. Je nachdem, ob das Fenster größer oder kleiner wird, müssen entweder die Flächen, in denen die Rollbalken vor oder nach der Größenveränderung liegen, invalidiert werden. **WindowGrow** macht sich die Sache etwas einfacher, indem sie immer sowohl vor wie auch nach der Größenveränderung die entsprechenden Gebiete mit **InvalGrow** markiert. **InvalGrow** selbst ist verhältnismäßig simpel:

```
PROCEDURE InvalGrow(fenster:  WindowPtr);
VAR
   r:          Rect;
   savePort:   GrafPtr;
BEGIN
   GetPort(savePort);
   SetPort(fenster);
   WITH fenster^.PortRect DO
      BEGIN
      SetRect(r,right-15,top,right,bottom);
      InvalRect(r);
          { rechten Rollbalken invalidieren }
      SetRect(r,left,bottom-15,right,bottom);
      InvalRect(r);
          { unteren Rollbalken invalidieren }
      END; { WITH fenster }
   SetPort(savePort);
END;{InvalGrow}
```

InvalGrow sichert zunächst den aktuellen GrafPort in einer lokalen Variablen und stellt ihn am Ende wieder her. Dies ist nötig, da es mit **InvalRect** nur möglich ist, Gebiete im aktuellen GrafPort (bzw. Fenster) zu invalidieren und deshalb das Fenster erst zum aktuellen GrafPort gemacht werden muß. Solange **InvalGrow** nur für das oberste Fenster aufgerufen wird, ist **fenster** sowieso schon der aktuelle GrafPort und das Sichern und Wiederherstellen des GrafPorts unnötig. In dieser Form ist **InvalGrow** aber allgemeingültiger und kann später auch anders verwendet werden.

Im wesentlichen berechnet **InvalGrow** nur einen jeweils 15 Punkte breiten Streifen am unteren und rechten Rand des Fensters und invalidiert diese.

8.4.3 Die Prozedur AenderFenster

AenderFenster führt programmabhängige Änderungen an einem Fenster aus, nachdem dessen Größe geändert wurde, abhängig von der neuen Größe. Man hätte diese Änderungen auch gleich in **WindowGrow** erledigen können, dies hätte aber zu einer Vermischung von programmabhängigem und unabhängigem Code geführt, der vermieden werden soll. Zudem kann **AenderFenster** in anderen Programmen auch wesentlich komplexer aussehen und schon deshalb eine eigenständige Prozedur rechtfertigen, während **WindowGrow** immer gleich aussieht.

```
PROCEDURE AenderFenster(fenster:  WindowPtr);
VAR
   r:           Rect;
   savePort:    GrafPtr;
BEGIN
   IF GetWKind(fenster) = grafKind THEN
   BEGIN
       GetPort(savePort);
       SetPort(fenster);
       WITH fenster^.PortRect DO
           BEGIN
           SetRect(r,left,top,right-15,bottom-15);
           ClipRect(r);
           END; { WITH fenster }
       SetPort(savePort);
       END;
END;{AenderFenster}
```

Wie alle programmabhängigen Prozeduren prüft **AenderFenster** zunächst, mit **GetWKind**, ob sie für dieses Fenster überhaupt "zuständig" ist. Falls ja, wird danach die **clipRgn** des betroffenen Fensters auf ein Rechteck gesetzt, das rechts und unten 15 Punkte — die Breite der Rollbalken — kleiner ist als das **portRect** des Fensters (bzw. des entsprechenden GrafPorts). Dadurch wird ein Überschreiben der Rollbalken oder des Grow-Icons durch eine im Fenster gezeichnete Grafik verhindert.

8.5 Zeichnen eines Fensters mit Grow-Icon

Damit wären die Änderungen am Programm **Rahmen0**, die es nun auch fähig machen, Fenster-Größen zu ändern, fast abgeschlossen. Es fehlen nur noch kleine Änderungen an den Routinen, die Fenster (auf einen **UpdateEvent** hin) zeichnen. Das Grow-Icon und die beiden leeren Rollbalken werden vom WindowManager ja nicht automatisch am Bildschirm dargestellt, obwohl es sich um ein Dokumentenfenster handelt, das zumindest immer ein Grow-Icon besitzt. Man kann dies entweder für einen Fehler in der ToolBox halten oder für eine Möglichkeit zur freien Gestaltung des Programms.

```
FUNCTION  ZeichneBild(bild: PictHandle;
                           wo: Point; scale: INTEGER);
BEGIN
   { wird unverändert aus Rahmen0 übernommen }
END;{ZeichneBild}

PROCEDURE ZeichneInhalt(fenster: WindowPtr);
VAR
   bei:     Point;
   r:       Rect;          { NEU ! }
   clip:    RgnHandle;     { NEU ! }
BEGIN
   FillRect(fenster^.portRect,white);
           { löscht den ungültigen Bereich }

   bei := Point(GetWRefCon(fenster));
   ZeichneBild(wPict,bei,scale);
           { frischt das gespeicherte Bild auf }


   { Der jetzt folgende Teil ist NEU ! }
   clip := NewRgn;
   GetClip(clip);
   ClipRect(fenster^.portRect);

   WITH fenster^.portRect DO
       BEGIN
       SetRect(r,right-15,top,right,bottom);
       FillRect(r,white);
       SetRect(r,left,bottom-15,right,bottom);
       FillRect(r,white);
       PenPat(black);
       PenMode(patCopy);
       DrawGrowIcon(fenster);
       END; { WITH fenster^.portRect }

   SetClip(clip);
   DisposeRgn(clip);
END;{ZeichneInhalt}
```

Obwohl die Rollbalken und das Grow-Icon nicht zum "eigentlichen" Inhalt des Fensters gehören, sondern eher eine Art Rahmen darstellen, faßt der WindowManager sie als dem Inhalt zugehörig auf. Fenster-Inhalt sind für ihn

alle Gebiete, für deren Aussehen er nicht selbst verantwortlich ist. Deshalb ist die richtige Stelle für die nötigen Änderungen die Prozedur **ZeichneInhalt,** die in **Rahmen0** (und allen folgenden Rahmenprogrammen) für das Auffrischen des Fensterinhalts verantwortlich ist.

Einen Großteil der Arbeit – nämlich das Zeichnen des Grow-Icons selbst – nimmt der WindowManager einem Programm auf Wunsch ab. Dazu muß nur die Prozedur **DrawGrowIcon** aufgerufen werden und ihr als Parameter das Fenster übergeben werden, dessen Grow-Icon gezeichnet werden soll. **DrawGrowIcon** zeichnet nicht nur das eigentliche Grow-Icon in der rechten unteren Ecke des Fensters, sondern deutet auch schon die Rollbalken durch entsprechende Linien am unteren und rechten Rand an. Diese leeren Rollbalken werden von Grow-Icon allerdings nicht vorher geleert (mit weiß gefüllt), weshalb **ZeichneInhalt** diese Aufgabe übernehmen muß.

Hauptaufgabe von **ZeichneInhalt** ist aber eine Korrektur der **clipRgn** vor dem Aufruf von **DrawGrowIcon.** In **AenderFenster** setzen wir die **clipRgn** des betroffenen Fensters ja gerade so, daß sie die Rollbalken und das Grow-Icon nicht mehr beinhaltet. Würde **ZeichneInhalt** **DrawGrowIcon** mit dieser Einstellung der **clipRgn** aufrufen, würde der Aufruf keinerlei Wirkung haben.

Deshalb sichert **ZeichneInhalt** zunächst die alte **clipRgn**, erweitert sie, so daß sie das ganze **portRect** des Fensters umschließt, und zeichnet erst dann das Grow-Icon und die Streifen für die Rollbalken. Hinterher wird die alte **clipRgn** natürlich wiederhergestellt.

8.6 Verbleibende Änderungen

Rahmen1 ist nun im wesentlichen fertig. Alle Änderungen, die an **Rahmen0** nötig waren, um auch Größenänderungen von Fenstern vom Anwendungs-Programm aus zu unterstützen, sind erfolgt. Es bleibt jedoch ein kleiner "Schönheitsfehler". Ein klein wenig "Programmkosmetik" muß deshalb noch erledigt werden.

Der bewußte Fehler ist nicht sehr offensichtlich und tritt vielleicht erst dann auf, wenn man ein wenig mit dem Programm experimentiert. Es handelt sich dabei allerdings nicht so sehr um einen "echten" Fehler als um eine Verletzung der Richtlinien, die Apple für MacProgramme aufgestellt hat und die die meisten Programme auf dem Macintosh auch einhalten.

Diese Richtlinie, die **Rahmen1** im Moment noch vernachlässigt, betrifft die Unterschiede zwischen der Darstellung der *Kontrollen* des vorne liegenden Fensters und hinten liegenden Fenstern. Kontrollen sind in diesem Zusammenhang bestimmte Gebiete, die auch grafisch als eigenständige Objekte hervorgehoben werden, in denen eine Maus-Klick eine genau definierte Wirkung hat, z.B. ein Fenster schließt oder dessen Inhalt rollt.

Kontrollen sollen immer nur beim obersten Fenster eine Wirkung haben, obwohl sie vielleicht auch bei teilweise verdeckten Fenstern — allerdings in anderer Form — sichtbar sind. So bewirkt ein Klick in das Grow-Icon eines hinten liegenden Fensters zunächst nur, daß es nach vorn kommt, und hat auf keinen Fall eine Größenänderung zur Folge. Die Kontrolle "Grow-Icon" ist bei einem hinten liegenden Fenster *inaktiv*. Um dies dem Benutzer klarzumachen, wird sie auch anders gezeichnet. Während bei einem aktiven Grow-Icon ein kleines Bildchen im entsprechenden Quadrat in der Fenster-Ecke erscheint, bleibt dieses Quadrat bei hinten liegenden Fenstern leer. Die folgende Abbildung zeigt, welches Aussehen des Grow-Icons für vorne liegende Fenster bzw. hinten liegende Fenster die Macintosh-Richtlinien festlegen:

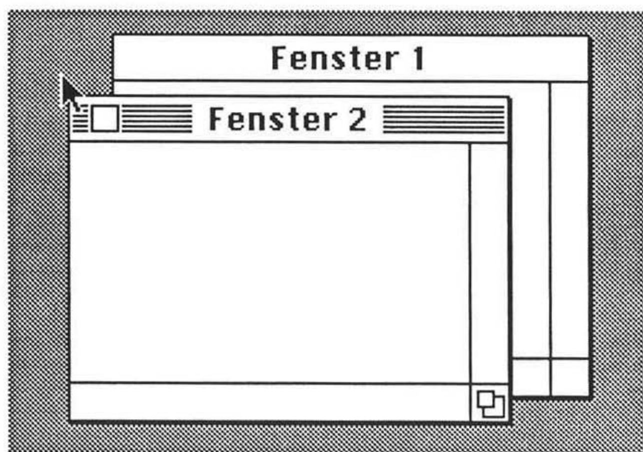


Bild 8 - 4: *Verschiedenes Aussehen von Grow-Icons*

Wie die Abbildung aber zeigt, garantiert **Rahmen1** im Moment ein solches Programmverhalten noch nicht, ja es kann sogar zum genau gegenteiligen Aussehen der beiden Grow-Icons kommen. Das des vordersten Fensters ist

leer, und das des hinten liegenden Fensters zeigt die beiden Quadrate, die ein Fenster-Wachstum andeuten.

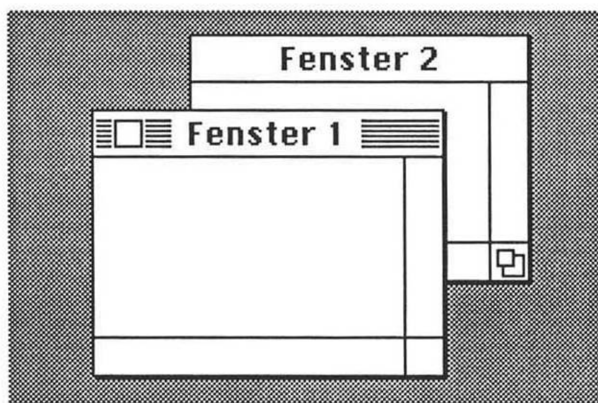


Bild 8 - 5: *Der letzte Programmfehler*

Dieser "Programmfehler" ist relativ schnell beseitigt, sobald erst einmal der Grund dafür klar ist: Wenn ein Fenster nach vorne kommt, erhält unser Programm für dieses Fenster einen Activate-Event und falls vorher ein anderes Fenster vorn lag, für dieses Fenster einen Deactivate-Event (es sind natürlich zwei Events desselben Event-Typs; **activateEvt**). Waren Teile des hinten liegenden Fensters von anderen Fenstern verdeckt, so wird dem Programm durch einen Update-Event mitgeteilt, daß diese Teile neu gezeichnet werden müssen. War jedoch kein Teil des hinten liegenden Fensters verdeckt — vor allem das Grow-Icon — so wird sein Inhalt auch nicht über einen Update-Event aufgefrischt, und das Grow-Icon behält sein altes Aussehen bei.

Dasselbe gilt natürlich erst recht für das Fenster, das an vorderster Stelle lag und nun nach hinten kommt. Es wird auf keinen Fall durch diesen Vorgang einen Update-Event erhalten, da ja kein Teil seines Inhalts dadurch freigelegt wird, daß es nach hinten kommt.

Die Lösung für dieses Problem liegt nun ganz einfach darin, dem Auffrischen der Fenster ein wenig "von Hand" nachzuhelfen. Jedesmal wenn ein Fenster einen Activate-Event erhält — es also zum vordersten Fenster wird oder diese Position an ein anderes Fenster abgibt — muß das Anwendungsprogramm dem WindowManager mitteilen, daß dadurch ein Teil des Fenster-Innern

ungültig geworden ist. Das Grow-Icon muß invalidiert werden, wozu sich natürlich die bereits für andere Zwecke geschriebene Prozedur **InvalGrow** anbietet. Die folgende Änderung wird deshalb an der Prozedur **WindowActivate** aus **Rahmen0** nötig:

```
PROCEDURE WindowActivate( fenster: WindowPtr;
                                   aktiv: BOOLEAN);
BEGIN
   IF aktiv THEN
      SetPort(fenster);
   IF (GetWKind (fenster) = grafKind) THEN
      InvalGrow(fenster);

   { Der Rest von WindowActivate bleibt gleich }
   .........

END;{WindowActivate}
```

Der Erfolg dieser kleinen Änderung kann am Beispielprogramm sofort ausgetestet werden, das nun unter den entsprechenden Umständen wirklich immer ein korrektes Aussehen der Grow-Icons, wie in Bild 8 – 4, zeigen sollte. Damit wären aber nun wirklich alle Änderungen abgeschlossen, die **Rahmen0** zu **Rahmen1** machen. In diesem Zustand unterstützt das Rahmenprogramm jetzt schon alle Standard-Verhaltensweisen, die jedes MacProgramm zeigen sollte, mit nur noch einer Ausnahme: den Schreibtisch-Utensilien.

8.7 Schlußbemerkung

Wie deutlich zu sehen war, ist das Ändern der Fenstergröße – und die damit verbundene Einführung der Kontrolle Grow-Icon – mit deutlich mehr Schwierigkeiten verbunden als z.B. das Verschieben von Fenstern oder die Unterstützung von PullDown-Menüs. In der vorliegenden Version ist Rahmen1 aber schon recht flexibel und kann z.B. relativ problemlos um jede Art von Objekten ergänzt werden, deren Position im Fenster abhängig von der Fenstergröße ist. Dies bildet auch schon eine sehr solide Grundlage für eine eventuelle Einführung von Rollbalken.

Ein sehr wichtiger Punkt des vorangegangenen Kapitels war die Pflege der **updateRgn** eines Fensters. In ähnlicher Weise, wie hier die Änderungen "beweglicher" Komponenten des Fenster-Inhaltes behandelt wurden, sollten

alle Änderungen des Fenster-Inhaltes gehandhabt werden. Tritt eine Änderung der dargestellten Abbildung ein, wird zunächst nur der entsprechende Bereich als ungültig markiert (invalidiert). Erst der dadurch veranlaßte Update-Event sorgt dann für das korrekte Aussehen der geänderten Flächen. "Direktes" Zeichnen in den Fenster-GrafPort, sobald eine Änderung eintritt, sollte nach Möglichkeit vermieden werden!

9 Resourcen – die zweite Ausbaustufe

Die erste Ausbaustufe des Rahmenprogramms kam noch mit den Informationen, die im ersten Teil des Buches vermittelt wurden, aus. Für die zweite Ausbaustufe muß jedoch etwas weiter ausgeholt werden. Der erste Teil dieses Kapitels wird sich mit den Grundlagen des Resource-Konzeptes des Macintosh beschäftigen, und erst im zweiten Teil wird dann das Rahmenprogramm entsprechend dieser neuen Kenntnisse modifiziert. Diese Änderungen sind nicht allzu umfangreich (aber schwerwiegend), weswegen der erste Teil des Kapitels im Verhältnis zum zweiten sehr umfänglich werden wird. Dies hat seinen Grund auch in der enormen Wichtigkeit des Resource-Konzeptes für die Programmierung des Macintosh, die mit der des Grafik-Pakets QuickDraw vergleichbar ist.

9.1 Grundlagen des Macintosh-Resource-Konzeptes

Der Macintosh konfrontiert fast jeden Programmierer mit neuen und gewöhnungsbedürftigen Konzepten für die Realisierung der Benutzer-schnittstelle von Programmen. Dies ist auch noch relativ naheliegend, wenn man diese neuartige Benutzerschnittstelle das erste Mal sieht. Andererseits bleiben aber auch andere Gebiete des Betriebssystems, mit denen ein Programm zu tun hat, vor Neuerungen nicht verschont.

Die Neuerung, mit der sich dieses Kapitel beschäftigt, sind Resource-Dateien, eine besondere Art von Dateien, die das Betriebssystem des Macintosh standardmäßig jedem Programm zur Verfügung stellt. Die geschickte Verwendung von Resource-Dateien kann ein Programm nicht nur vollkommen unabhängig von der Muttersprache seiner Benutzer machen, sondern kann auch die Programmierung an vielen Stellen vereinfachen.

9.1.1 Ursprünge des Resource-Konzeptes

Der Macintosh war von Anfang an als "internationaler Computer" gedacht gewesen. Seine Hardware und auch die (im ROM) eingebaute Software sollten soweit als möglich unabhängig von seinen amerikanischen Erbauern und an nationale Gegebenheiten anpaßbar sein. Auf der Hardware-Seite hat sich diese Prämisse z.B. darin niedergeschlagen, daß sich auf dem gesamten Gehäuse des Macintosh keinerlei Texte, außer gesetzlich vorgeschriebenen Beschriftungen und dem Namenszug "Macintosh", befinden, sondern nur kleine Bilder (Icons). Die Tastatur ist problemlos auswechselbar und wird softwareseitig durch eine Betriebssystemkomponente dekodiert, die erst beim Start des Computers von einer Diskette geladen wird – aus einer Resource-Datei übrigens.

Der erste Schritt, auch die Programme nationalitätsunabhängig zu machen, bestand darin, alle Texte, die ein Programm benötigt und der Benutzer zu sehen bekommt, getrennt vom eigentlichen Programm zu speichern. In einem Programm würde man also nicht mehr schreiben dürfen:

```
writeln ('Hallo, hier ist der Macintosh');
```

Die korrekte Art und Weise, so etwas "international" zu machen, sieht etwa aus wie folgt:

```
HoleMeldung(einString,2519);
writeln(einString);
```

Dabei würde **HoleMeldung** den eigentlichen Text für die Meldung aus einer besonderen Datei einlesen.

Sehr schnell war aber klar, daß es auf einem Computer mit einer stark grafisch orientierten Benutzerschnittstelle, wie dem Macintosh, nicht ausreichen würde, nur die Texte vom Programm zu trennen. Ein großes Problem in dieser Hinsicht ist allein schon die unterschiedliche Länge von Meldungen gleichen Sinns in unterschiedlichen Sprachen. So sind englische Meldungen im Durchschnitt um ein Drittel kürzer als die entsprechenden deutschen. Bildschirm-Formulare, die in der englischen Fassung korrekt aussehen, können total unübersichtlich werden, wenn man nur ihre Texte eindeutscht und nicht gleichzeitig auch die Positionen der Texte entsprechend ändert.

Bald war den Designern des Macintosh klar, daß nach Möglichkeit alles, was der Benutzer zu sehen bekommen sollte, einschließlich Texten, Grafiken,

Icons und deren Positionen auf dem Bildschirm, in vom Programm getrennten Dateien gelagert werden sollte. Aus dieser Idee entstand das Konzept der Resource-Dateien.

Dieses Konzept wird von der größten Anzahl der Macintosh-Programme auch genutzt. Die Übertragbarkeit von Macintosh-Software auf anderssprachige Märkte ist deshalb relativ hoch. So ist es im Extremfall mit Hilfe von Resourcen durchaus möglich, ein komplettes Programm aus dem Deutschen ins Italienische zu übersetzen, ohne auch nur ein Byte des Programms zu ändern oder auch nur einen Blick auf den ursprünglichen Sourcetext des Programms zu werfen. (Der Nutzen hat sich allerdings in dieser Hinsicht als nicht ganz so hoch erwiesen, wie sich das die Entwickler vorgestellt hatten. Es genügt eben nicht, ein Programm zu übersetzen, um es in einem anderen Land zu verkaufen; ein passendes Handbuch und eine geänderte Verpackung gehören oft auch dazu und machen üblicherweise mehr Arbeit als die Übersetzung des Programms selbst.)

Um die Zusammengehörigkeit zwischen einem Programm und seinen Resourcen hervorzuheben (und auch, um zu verhindern, daß ein Programm aus vielen unterschiedlichen Dateien besteht, die auf verschiedenen Disketten "herumschwirren"), sind die Resource-Dateien nicht als separate Dateien konzipiert, sondern als ein besonderer Teil einer Datei.

Jede Datei auf dem Macintosh gabelt sich auf in zwei Teile oder Gabeln (in *Inside Macintosh* auf englisch *forks* genannt): einen Daten-Teil und einen Resource-Teil (in IM *data fork* bzw. *resource fork*). Beide Teile zusammen bilden eine Datei, die der Finder auch als ein Icon anzeigt.

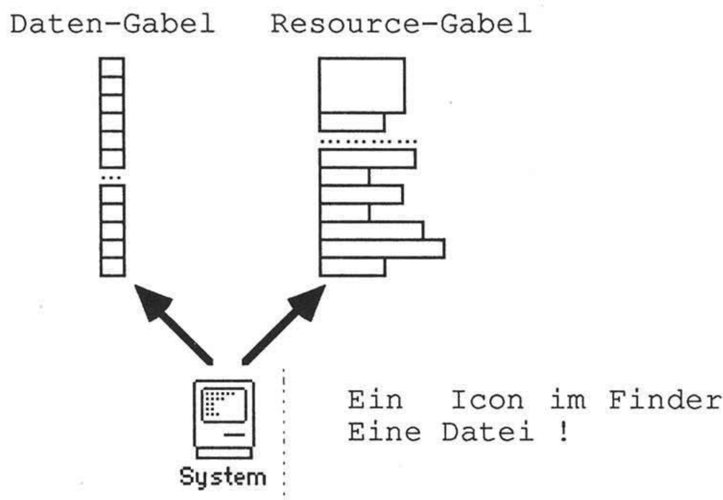


Bild 9 - 1:*Die zwei Gabeln einer Datei*

Beide Teile werden vom Betriebssystem völlig getrennt verwaltet, als wären es zwei unterschiedliche Dateien, und es gibt auch zwei getrennte Pakete in der ToolBox, mit denen die beiden unterschiedlichen Gabeln bearbeitet werden. Es handelt sich ja auch um völlig unterschiedliche Arten von Daten, die in den beiden Teilen gespeichert werden. Während für die Handhabung des Daten-Teils einer Datei der FileManager zuständig ist, der ähnliche Operationen enthält wie vergleichbare Dateiverwaltungen auf anderen Rechnern auch, gibt es für den Resource-Teil einer Datei den ResourceManager, der vollkommen andersartige Operationen zur Verfügung stellt.

Resourcen haben mit der Zeit nicht nur für die internationalen Aspekte von Software Verwendung gefunden, sondern noch für eine Vielzahl anderer Anwendungen. Viele Programmierer fanden die Operationen, die der ResourceManager anbietet, geeigneter für ihre Anwendung als die normalen Operationen der Datei-Verwaltung (des FileManagers).

Viele Anwendungen nutzen allerdings nur eine Gabel einer Datei, die andere bleibt leer. Einige Programme legen z.B. alle ihre Daten nur in die Daten-Teile der entsprechenden Dateien, andere nutzen nur den Resource-Teil und

lassen den Daten-Teil leer. Optimal ist allerdings meist eine ausgewogene Nutzung beider Teile; je nach der Strukturierung der von einem Programm verwalteten Daten. Finder und Betriebssystem des Macintosh nutzen zumeist beide Teile ihrer Dateien, besonders intensiv aber den Resource-Teil.

Prinzipiell scheint die folgende Faustregel ganz sinnvoll zu sein: Sind die gespeicherten Daten sehr homogen (d.h. alle Datensätze sind von einem Typ) und/oder sind alle Datensätze in einer Datei von gleicher Länge, so ist es effizienter, diese Datensätze im Daten-Teil abzuspeichern. Werden hingegen eine Vielzahl unterschiedlicher Datensätze mit unterschiedlichen Längen verwaltet, so bietet sich dafür die Resource-Gabel einer Datei an.

Zur Handhabung solcher Situationen, die verschiedene Datensätze variabler Länge verlangen, werden üblicherweise auf anderen Rechnern mehrere verschiedene Dateien und/oder aufwendige Datenbank-Pakete eingesetzt oder selbst geschrieben. Auf dem Macintosh ist es dank des Resource-Konzeptes oft möglich, all diese unterschiedlichen Dateien zu einer zusammenzufassen und auch ohne zusätzliche Datenbank-Software recht anspruchsvolle Datenverwaltungen zu realisieren.

Weil sich diese Nützlichkeit des Resource-Konzepts ziemlich früh herausgestellt hat, wird es auch von anderen Teilen der ToolBox intensiv verwendet. Dies ist für den Benutzer dieser anderen Pakete nicht unbedingt immer ersichtlich oder notwendig zu wissen. Wer sich aber eingehender damit beschäftigt, wird bald feststellen, daß es fast keinen größeren Teil der ToolBox gibt, der nicht in irgendeiner Weise Resourcen nutzt. Das Verständnis des Resource-Konzeptes ist deshalb, wenn doch nicht unbedingt notwendig, so doch sehr nützlich für ein tieferes Verständnis der ganzen ToolBox.

9.1.2 Eigenschaften einer einzelnen Resource

Der Resource-Teil einer Datei hat seinen Namen daher, daß er einzelne *Resourcen* enthält. Trotz ihres etwas hochtrabenden Namens ist eine *Resource* allerdings im wesentlichen nichts anderes als ein zusammenhängender Datenblock variabler Länge.

Jede Resource einer Resource-Gabel muß durch maximal drei Merkmale eindeutig identifiziert sein. Es darf keine zweite Resource geben, bei der alle drei Merkmale übereinstimmen. Diese drei Merkmale sind:

Der **Resource-Typ** ist eine aus vier Buchstaben bestehende Kennung, die die verschiedenen Arten von Resourcen innerhalb des Resource-Teils einer Datei unterscheidet. Wichtige Resource-Typen sind z.B. 'STR ' (die Leerstelle als vierter Buchstabe ist signifikant), 'MENU' und 'WIND'. Die entsprechenden Resourcen enthalten Texte (für Meldungen etc.) sowie die Beschreibungen für PullDown-Menüs und Fenster.

Diese Typ-Bezeichnungen können nicht nur Großbuchstaben enthalten, sondern auch Kleinbuchstaben ('MENU', 'Menu' und 'menu' sind vier unterschiedliche Typen), obwohl es üblich ist, Großbuchstaben dafür zu nehmen. Intern können diese vier Buchstaben (vier Bytes) als eine 32–Bit-Zahl abgespeichert werden, und für den ResourceManager ist der Typ auch nur eine Zahl (die ja von den meisten Computern viel leichter zu handhaben ist als ein Text).

Der **Resource-Schlüssel** oder **Resource-ID** unterscheidet Resourcen des gleichen Typs voneinander. Es handelt sich dabei um eine ganze Zahl zwischen –32.768 und 32.767 (eine 16–Bit-Zahl). Dabei sind allerdings alle negativen Werte und die Zahlen bis einschließlich 127 für das System reserviert. Programme sollten ihren Resourcen Schlüssel zwischen 128 und 32.767 geben.

Jede Resource hat einen Typ und einen Schlüssel; zusammen identifizieren die beiden eindeutig jede Resource einer Resource-Gabel. Zwei Resourcen gleichen Typs können also nicht denselben Schlüssel haben — Resourcen unterschiedlichen Typs können allerdings gleiche Schlüssel haben, was oft sogar sehr praktisch ist.

Die dritte Kennzeichnung einer Resource dient mehr der Bequemlichkeit des Programmierers und der Lesbarkeit eines Programms. Neben dem Schlüssel kann jede Resource noch einen **Namen** haben, der aus 1 bis 255 Buchstaben bestehen kann (es handelt sich um einen Pascal-String). Zwei Resourcen gleichen Typs können wieder nicht denselben Namen haben, während gleiche Namen bei Resourcen unterschiedlichen Typs möglich und oft sinnvoll sind. Resource-Namen sind optional: während jede Resource einen Schlüssel haben muß, ist ein Name nicht zwingend vorgeschrieben.

Der ResourceManager und der MemoryManager arbeiten eng zusammen. Resourcen erlauben eine simple Form der *virtuellen* Speicherverwaltung, bei der Daten sich nur dann im Speicher befinden, wenn sie benötigt werden, und auf die Diskette ausgelagert werden, während sie nicht benötigt werden. Jedesmal, wenn ein Programm eine Resource benötigt, so fordert es diese (identifiziert durch Typ und Schlüssel oder Namen) vom ResourceManager

an und erwartet daraufhin vom ResourceManager einen Handle, der auf den eingelesenen Datenblock der Resource verweist.

Um dem Programm diesen Handle liefern zu können, prüft der ResourceManager zunächst seine internen Verzeichnisse, ob diese Resource vielleicht schon früher einmal benötigt wurde und sich deshalb noch im Speicher befindet. Wenn ja, wird der entsprechende Handle festgestellt und an das aufrufende Programm zurückgegeben. Wenn nein, wird ein verschiebbarer Datenblock im Heap angelegt, in den dann der Datenblock der Resource von der Diskette eingelesen wird. Ein Handle auf diesen neuen Block im Heap wird dann an das Programm zurückgegeben. Die Eigenschaften des Handles, ob er z.B. **locked** oder **purgeable** ist (vergleiche das Kapitel *Speicher-Verwaltung*), bestimmen Informationen, die der ResourceManager zusätzlich zu jeder Resource verwaltet.

An dieser Stelle kann nun auch endlich der Nutzen des Flags **purgeable** eines verschiebbaren Datenblocks erläutert werden, der im Kapitel *Speicher-Verwaltung* nur kurz angeschnitten werden konnte. Das Flag **purgeable** wird fast nur für verschiebbare Blocks verwendet, in denen sich vom ResourceManager eingelesene Resourcen befinden. Ist dieser Block **purgeable**, so kann er, sobald der freie Platz im Heap knapp wird, gelöscht (ge*purgt*) werden. Vorher wird dem ResourceManager allerdings noch mitgeteilt, daß die Absicht besteht, diesen Block zu purgen. Dadurch erhält dieser die Gelegenheit, nachzuschauen, ob sich die Daten innerhalb des entsprechenden Blocks geändert haben, und sie auf der Diskette zu sichern, bevor der Block gelöscht wird.

Die Verwendung dieser Fähigkeit des ResourceManagers sollte allerdings mit äußerster Vorsicht geschehen. Jeder Handle, der auf einen Block verweist, der **purgeable** ist, sollte vor jeder Verwendung darauf überprüft werden, ob sich die Daten an seinem Ende überhaupt noch im Speicher befinden. Nur bei größeren Resourcen, die zudem nur relativ sporadisch benötigt werden, sollte man das **purgeable**-Flag nutzen.

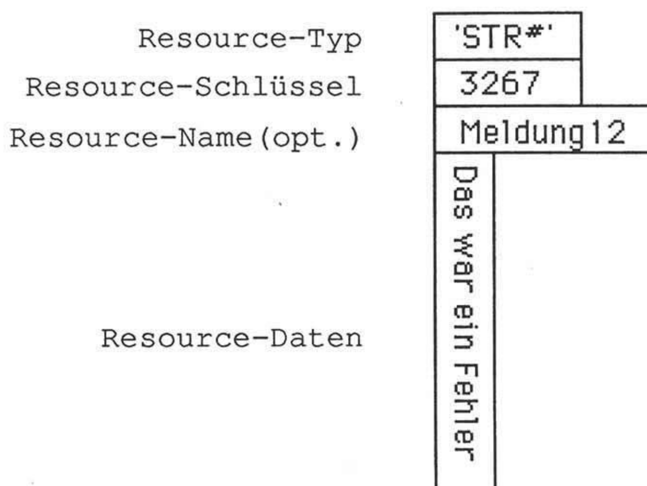


Bild 9 - 2: *Aufbau einer einzelnen Resource*

Eine vollständige Liste der möglichen Eigenschaften einer Resource folgt nun (es handelt sich um eine Reihe von Flags, die fast alle für die Zusammenarbeit von ResourceManager und MemoryManager gedacht sind.):

• **resPurgeable** wurde bereits beschrieben. Es entspricht dem Flag **purgeable** des entsprechenden Handles, sobald die Resource das erste Mal in den Speicher gelesen wurde. (Nach dem ersten Einlesen kann das **purgeable**-Flag des Handles mit den entsprechenden Operationen des MemoryManagers jederzeit geändert werden.)

• **resLocked** bestimmt ebenfalls das entsprechende Flag des Handles, das die Resource-Daten nach dem Einlesen enthält. (Dadurch wird dafür gesorgt, daß der entsprechende Datenblock nicht verschiebbar im Heap ist, obwohl er an einem Handle hängt – eine bei Resourcen selten benötigte Möglichkeit.)

• **resSysHeap** bestimmt, in welchen Heap der ResourceManager den verschieblichen Datenblock beim Einlesen der Resourcedaten legt. Ist das Flag gesetzt, so liest er es in den relativ kleinen System-Heap ein. Üblich ist für die meisten Resourcen eines Anwendungs-Programms, daß sie in den Programm-Heap eingelesen werden.

• **resProtected** verhindert, daß gewisse Aspekte der entsprechenden Resource geändert werden. Ist dieses Flag gesetzt, kann diese Resource nicht aus ihrer Datei entfernt werden, und ihr Name und ihr Schlüssel können nicht geändert werden.

• **resPreload** sorgt dafür, daß Resourcen, bei denen dieses Flag gesetzt ist, sofort in den Heap eingelesen werden, sobald die entsprechende Resource-Datei geöffnet wird. Dies kann für Resourcen, die sofort nach dem Öffnen der Datei (z.B. beim Start eines Programms) benötigt werden, ganz praktisch sein, da das Diskettenlaufwerk dann gar nicht erst zur Ruhe kommt.

• **resChanged** bedeutet, daß die entsprechende Resource geändert wurde, seit sie das letzte Mal von der Diskette eingelesen wurde. Bevor der Datenblock, der diese Resource im Hauptspeicher enthält, gepurgt wird oder die Resource-Gabel ihrer Datei geschlossen wird, muß diese Resource also auf die Diskette zurückgeschrieben werden. **resChanged** hat also nur eine Bedeutung für Resourcen, die sich gerade im Speicher befinden und wird ansonsten ingnoriert.

Die Möglichkeiten, die sich aus den verschiedenen Eigenschaften (Flags) einer Resource ergeben, werden allerdings vom Rahmenprogramm in keiner Weise genutzt; sie wurden hier nur der Vollständigkeit halber erwähnt. Sie spielen natürlich in jedem echten Programm eine große Rolle – insbesondere auf dem kleinen Macintosh, auf dem die Möglichkeit der virtuellen Speicherverwaltung oft intensiv genutzt werden muß.

9.1.3 Aufbau des Resource-Teils einer Datei

Während der Daten-Teil einer Datei für das Betriebssystem eigentlich nur aus einem amorphen Srom von Bytes besteht, der keine Struktur besitzt, kann der Resource-Teil recht komplexe Strukturen aufweisen. Die Einheiten, aus denen sich die Resource-Gabel zusammensetzt, sind einzelne *Resourcen*, eigenständige Datenblöcke variabler Länge. Zusätzlich zu den "nackten" Daten, die eine Resource enthält, besitzt sie noch eine Reihe von Eigenschaften, die im letzten Abschnitt bereits erläutert wurden. Diese Eigenschaften werden, zusätzlich zu den Daten selbst, ebenfalls in der Resource-Gabel verwaltet.

Ein Programm, das den Inhalt des Datenblocks einer Resource erfahren will, braucht sich nicht um die Position dieses Blocks innerhalb der Datei, bzw. deren Resource-Teil, zu kümmern und muß auch nicht die Länge dieses Blocks kennen. Es muß diese Resource nur gegenüber dem

ResourceManager genau identifizieren. Daraufhin liest dieser die Daten und sämtliche Merkmale ein und gibt dem Programm einen Handle auf den Datenblock zurück. Diese Identifizierung besteht entweder aus dem Typ und dem Schlüssel oder aus dem Typ und dem Namen der Resource.

Für das Programm erscheint der Resource-Teil einer Datei wie eine Ansammlung der einzelnen Resourcen, die keine Reihenfolge und Position kennt. In dieser Hinsicht ist eine Resource-Gabel einem Heap sehr ähnlich, in dem Datenblöcke über Handles verwaltet werden. Wie bei einem Handle kümmert es ein Programm bei einer Resource nicht, wo diese gespeichert ist, wie es diesen Datenblock findet oder wie er verwaltet wird. All dies kann der Speicherverwaltung bzw. in diesem Fall dem ResourceManager überlassen werden.

Intern verwaltet der ResourceManager natürlich schon eine Reihe von Informationen über die Position der Resourcen in einer Datei. Physikalisch ist auch die Resource-Gabel einer Datei nur eine Folge von Bytes. In dieser Folge werden die einzelne Resourcen (plus einige Verwaltungsinformationen) einfach hintereinander abgespeichert. An das Ende der Resource-Gabel einer Datei hängt der ResourceManager stets die **Resource-Karte** (in IM *resource map*), in der verzeichnet wird, wo welche Resource relativ zum Beginn der Byte-Folge, die den Resource-Teil einer Datei bildet, steht. Diese Resource-Karte entspricht in etwa den Indexstrukturen einer Datenbank auf anderen Rechnern. Sie enthält selbst keine Daten (einige schon, aber ihre Hauptaufgabe ist nicht die Datenspeicherung), sondern ist nur beim Auffinden der eigentlichen Daten behilflich.

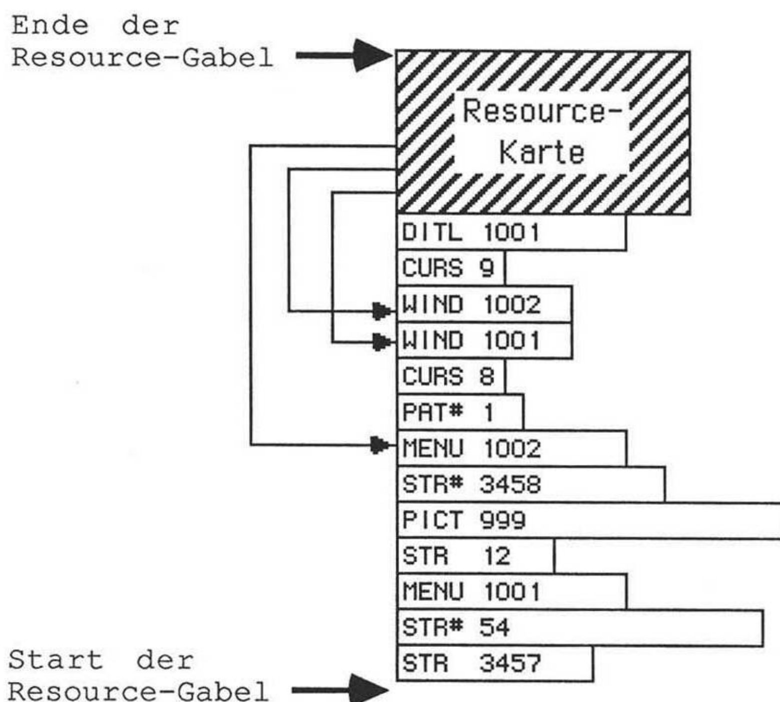


Bild 9 - 3: *Aufbau des Resource-Teils einer Datei*

Sobald die Resource-Gabel einer Datei geöffnet wird (für das Öffnen von Resource-Gabel und Daten-Gabel sind unterschiedliche Operationen zuständig), liest der ResourceManager die gesamte Resource-Karte in den Speicher. Dies bringt zwei entscheidende Vorteile mit sich. Einmal wird die Suche nach einer bestimmten Resource natürlich wesentlich schneller, da sie sich ja im schnellen Hauptspeicher abspielt. Erst wenn feststeht, wo sich der Datenblock der Resource befindet, braucht auf die viel langsamere Diskette oder Festplatte zugegriffen werden.

Andererseits werden Änderungen an Resourcen dadurch wesentlich leichter, daß die Resource-Karte sich im Hauptspeicher befindet. Werden einzelne Resourcen z.B. länger, so wird auch die Resource-Gabel länger (und überschreibt eventuell die alte Resource-Karte auf der Diskette). Diese Änderungen werden aber zunächst nur in der Resource-Karte im

Hauptspeicher verzeichnet. (Nach dem Öffnen einer Resource-Gabel wird die alte Resource-Karte auf der Diskette nicht mehr benötigt.) Erst wenn alle Änderungen abgeschlossen sind und die Resource-Gabel einer Datei wieder geschlossen ist, wird die Resource-Karte aus dem Speicher zurück auf die Diskette geschrieben — hinter alle Resourcen, die sich zu diesem Zeitpunkt in der Resource-Gabel befinden.

Während des Ablaufs der meisten Programme werden normalerweise nur zwei Resource-Gabeln geöffnet sein: die Resource-Gabel der Datei "System", die immer offen ist, und die des Programmes selbst, die unter anderem auch den Programm-Code selbst enthält. (Jede <u>Programm</u>-Datei enthält eine Resource-Gabel!) Das Programm kann allerdings jederzeit weitere Resource-Dateien öffnen und hat dann auch Zugriff auf die in diesen enthaltenen Resourcen. Hierbei kann es unter Umständen dazu kommen, daß in zwei verschiedenen Dateien zwei Resourcen gleichen Typs mit gleichem Schlüssel vorkommen. Auch in diesem Fall muß das Verhalten des ResourceManagers berechenbar bleiben, d.h., es muß stets definiert sein, welchen Handle der ResourceManager zurückgibt, wenn eine Resource angefordert wird.

Hierzu ist immer eine Reihenfolge definiert, an die sich der ResourceManager bei der Suche nach Resourcen hält. Im Normalfall durchsucht er zunächst die Resource-Karte der zuletzt geöffneten Resource-Gabel nach den gewünschten Resourcen und dann die "älteren" Resource-Karten. Dies sollte man beim Öffnen von Resource-Dateien immer im Auge behalten. Resourcen in neu geöffneten Dateien "überdecken" immer ältere Resourcen. Deshalb sollte stets vermieden werden, seinen Resourcen Nummern oder Namen zu geben, die für das System reserviert sind!

9.1.4 Erzeugung und Modifikation von Resourcen

Damit ein Programm überhaupt Resourcen nutzen kann, müssen diese natürlich zunächst einmal irgendwie in die entsprechenden Resource-Dateien "hineingelegt" werden. Dies könnte natürlich vom Programm selbst aus mit den Operationen des ResourceManagers geschehen, was aber nicht die Idee hinter den Resourcen ist, die ein Programm ja unabhängig vom genauen Inhalt seiner Resourcen machen sollen. Ein Programm sollte ja nur wissen, welche Resourcen es nutzen kann, aber nicht, was diese genau enthalten!

Deshalb hat Apple mehrere Werkzeuge entwickelt, um Resourcen zu erzeugen bzw. zu modifizieren, die sich in einer Programm-Datei oder in separaten Resource-Dateien befinden. Diese zerfallen in zwei größere Gruppen: Resource-Compiler und Resource-Editoren.

Resource-Compiler lesen eine Text-Datei, die Beschreibungen von verschiedenen Resourcen enthält, und bilden aus diesen textuellen Beschreibungen die entsprechenden Datenblöcke, die dann in eine neue oder bereits vorhandene Resource-Gabel einer Datei geschrieben werden. Ein inzwischen recht weit verbreiteter Resource-Compiler ist der "RMaker", den es sowohl auf der Lisa, im Pascal-Programm-Entwicklungssystem, wie auch auf dem Macintosh gibt. Die Formate für die Beschreibungen einiger wichtiger Resource-Typen des RMakers folgen im nächsten Abschnitt.

Resource Compiler

Source File Rahmen.R	**Output File** Rahmen
TYPE WIND ,1001 Fenster 1 100 100 300 300 Visible GoAway 0 0 ,1002 Fenster 2 110 110 310 310 InVisible GoAway 0 0 TYPE STR# ,1001 1 "STR "-Resource	**Data Size:** 3442 **Map Size:** 2218 **Total Size:** 5660

Bild 9 - 4: *Ein Resource-Compiler (RMaker) in Aktion*

Resource-Compiler haben jedoch immer auch die bekannten Nachteile aller Compiler. Das Ergebnis entsteht aus einer manchmal recht kryptischen Beschreibung, der man das Endergebnis nicht immer sofort ansieht. Viele Versuche, bevor das erwünschte Aussehen einer Resource endlich erreicht ist, sind die Regel. (Z.B. müssen die Koordinaten für die Ecken eines Fensters als Zahlen angegeben werden. Nicht jeder kann sich aus diesen

Zahlen sofort die Größe und die Position des Fensters am Bildschirm vorstellen.)

Eine Möglichkeit, die der sonstigen komfortablen Bedienung des Macintosh eher entspricht, sind die Resource-Editoren. Sie erlauben eine **interaktive** Erzeugung bzw. Modifikation von Resourcen, wobei man das Endergebnis seiner Arbeit sofort am Bildschirm betrachten kann. Es gibt inzwischen, aus den verschiedensten Quellen, eine ganze Reihe kleinerer Resource-Editoren, mit denen man nur einige wenige Resource-Typen bearbeiten kann ("MenuEdit", "DialogEdit" etc.). Dazu kommen zwei größere Versionen, die von Apple selbst stammen und die Bearbeitung einer großen Anzahl von Resource-Typen erlauben ("REdit" und "ResEdit"). Diese sind im Moment noch nicht ganz fertig, sollten aber zum Zeitpunkt, da Sie dies lesen, in ihren endgüligen Versionen verfügbar sein.

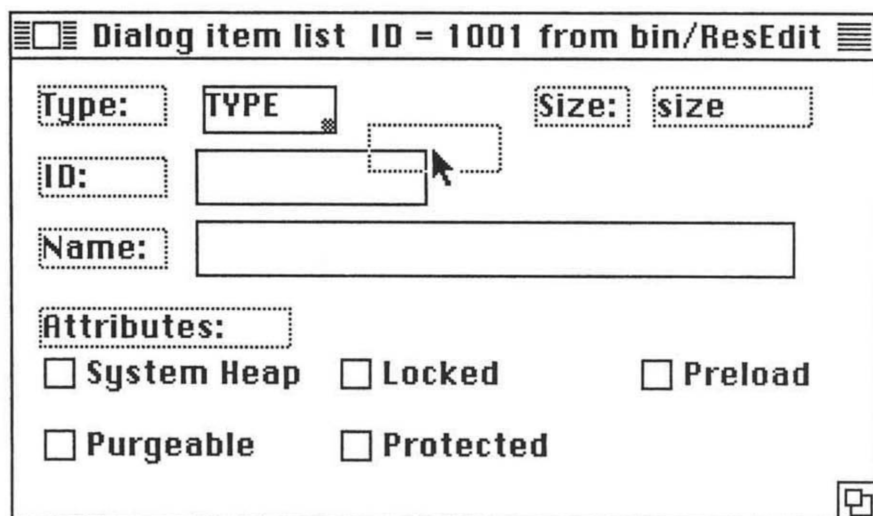


Bild 9 - 5: *Bearbeitung von Resourcen mit einem Resource-Editor (ResEdit)*

Resource-Editoren erlauben beispielsweise das Verschieben von Texten, Knöpfen und Bildern innerhalb von Dialogen mit der Maus, die Positionierung von Fenstern am Bildschirm direkt mit der Maus, das Ändern von Zeichensätzen, etc. Sie haben auch gewisse kleinere Nachteile gegenüber einem Resource-Compiler, sind diesem aber im allgemeinen vorzuziehen. Da

der RMaker aber im Moment große Bedeutung hat, und zweifellos für viele Anwendungen auch behalten wird, werde ich auf ihn im nächsten Abschnitt noch näher eingehen.

9.1.5 Die wichtigsten Resource-Typen

Einige der gebräuchlichsten Resource-Typen sind in jeden Resource-Compiler und Resource-Editor sozusagen "festeingebaut". Einige davon sollen nun beschrieben werden. Ich werde bei dieser Beschreibung besonderen Wert auf die Beschreibung des Eingabe-Formates für den *RMaker* legen. Dieses Eingabe-Format enthält nahezu alle wichtigen Aussagen über den Inhalt und den Aufbau einer Resource und wird für die Programmentwicklung mit den meisten Programmiersprachen im Moment noch benötigt. Eine ausführliche Beschreibung der Formate der einzelnen Resource-Typen findet sich natürlich in der Dokumention des RMakers.

Das Vorgehen zur Erzeugung derselben Resource-Typen mit einem Resource-Editor ist meist vergleichweise einfach und intuitiv, weswegen es hier nicht besonders dargestellt wird.

Einige Resource-Typen werden hier noch nicht vorgestellt, da sie erst zusammen mit einigen weiteren Erläuterungen in den folgenden Kapiteln verständlich werden.

9.1.5.1 Allgemeine Formatangaben

Eine Eingabe-Datei für den RMaker hat eine recht simple Struktur. Nach einigen Angaben über die zu erzeugende Resource-Datei als Ganzes (z.B. ihren Namen) folgt eine Liste von Resourcen, die üblicherweise nach Resource-Typen gruppiert ist. Jede Gruppe beginnt mit der Festlegung des Typs der folgenden Resourcen in folgender Form:

TYPE *XXXX*

Die Buchstaben XXXX sind dabei natürlich durch den entsprechenden Typ zu ersetzen (z.B. "**TYPE** STR#"). Wichtig ist, daß alle vier Buchstaben notwendig und signifikant für die Identifizierung des Typs sind. Ist eine gewählte Bezeichnung kürzer als vier Buchstaben, so muß der Rest mit Leerstellen aufgefüllt werden.

Hinter der Zeile, die den Typ festlegt, kommt eine Liste der Beschreibungen der einzelnen Resourcen des ensprechenden Typs. Jede dieser Resource-Beschreibungen beginnt mit einem Beschreibungs-Kopf: einer Zeile, die den Namen und den Schlüssel der folgenden Resource festlegt. Name und Schlüssel sind durch ein Komma getrennt, und der Name kann auch entfallen. Die Beschreibung der Resource-Daten folgt in den nächsten Zeilen (und ist natürlich vom Typ abhängig). Die entsprechenden Zeilen einer Eingabedatei für den RMaker könnten z.B. aussehen wie folgt:

```
TYPE STR
FehlerMeldung,531
Das war wohl nix !
```

Hinter dem Schlüssel der Resource kann in Klammern auch noch eine Zahl stehen, die die Attribute der fertigen Resource spezifiziert. Wie diese Zahl genau zu verstehen ist, wird im folgenden Abschnitt bei der Beschreibung der Prozedur **SetResAttrs** näher erläutert. Sie entspricht dem zweiten Parameter von **SetResAttrs**. Eine Resource-Beschreibung mit Attribut-Angabe sähe dann z.B. aus wie folgt:

```
TYPE STR
FehlerMeldung,531(32) ; <- Resource ist purgeable
Das war wohl nix !
```

Der Name einer Resource kann auch fortfallen, dann besteht der Kopf der Resource-Beschreibung nur aus einem Komma, gefolgt vom Schlüssel und eventuell einer Attribut-Spezifikation. (Das Komma ist zwingend vorgeschrieben, auch wenn kein Name vorhanden ist!)

```
TYPE STR
,531(32) ; <- Resource ohne Namen
Das war wohl nix !
```

Damit wäre die Beschreibung des gemeinsamen Formats aller Resource-Typen abgeschlossen. Der RMaker ist in seiner Behandlung der Eingabe nicht gerade robust zu nennen und prüft deren Plausibilität kaum nach. Deshalb ist es auf alle Fälle guter Stil, hinter den Angaben zu jeder einzelnen Resource eine Leerzeile zu lassen. Einige Resourcen, deren Angaben keine feste Länge haben, benötigen diese Leerzeile sowieso. Vor jedem neuen Typ sollte auch eine zusätzliche Leerzeile eingefügt werden, um dem RMaker

"eine Chance zu geben", das Ende der Resourcen vom letzten Typ auch auf alle Fälle zu erkennen.

9.1.5.2 Die gebräuchlichsten Resource-Typen

Der Rest dieses Abschnitts beschreibt das Format der gebräuchlichsten Resource-Typen in Form von "Formularen". Diese Formulare tragen in ihrer ersten Zeile stets die Typ-Angabe für den Resource-Compiler in der Form:

TYPE *typ*(4 Buchstaben)

Dahinter kommt das Format für die Beschreibung einer einzelnen Resource des entsprechenden Typs. Sie beginnt natürlich immer mit der Kopfzeile:

ResName,*ResID*(*ResAttr*)

Darunter folgt dann die Beschreibung der eigentlichen Resource-Daten. Genau wie bei der Beschreibung der Kopfzeile, tauchen in der Formatbeschreibung kursiv geschriebene Bezeichner auf, die in einer wirklichen Resource-Beschreibung gegen eine Zahl oder einen Text ersetzt werden müssen. Fettgedruckte Bezeichner stehen für einen Text, der genau so in der Beschreibung erscheinen muß. Stehen zwei oder mehr fettgedruckte Bezeichner in geschweiften Klammern, so ist an dieser Stelle eine Auswahl zu treffen. Einer der Bezeichner in der Klammer muß an dieser Stelle in der wirklichen Resource-Beschreibung auftauchen (ohne die geschweiften Klammern natürlich).

Aber nun zu den wirklichen Format-Beschreibungen:

TYPE STR
ResName ,*ResID* (*ResAttr*)
Text

Der Resource-Typ 'STR ' — diese Buchstaben-Kombination steht natürlich für "String" — ist einer der simpelsten Typen überhaupt. Das Format der Resource-Daten besteht aus einem Byte, das die Anzahl der folgenden Buchstaben enthält und danach den Buchstaben selbst. (Es können zwischen 0 und 255 sein.) Der RMaker erzeugt die entsprechende Resource aus *Text*, wobei der gesamte Inhalt dieser Zeile (bis max. 255 Buchstaben) in die Resource-Daten gelangt.

```
TYPE STR#
ResName ,ResID (ResAttr)
AnzahlStrings
String1
String2
.....
```

Resourcen vom Typ 'STR#' (String-Liste) enthalten eine Folge von Strings, deren Format ansonsten dasselbe ist wie beim Resource-Type 'STR '. Zuerst kommt eine 16–Bit-Zahl, die die Anzahl der folgenden Strings enthält und danach den Strings selber, die einfach hintereinander gelegt werden. Die Resource-Beschreibung beginnt entsprechend mit der Anzahl der folgenden Zeilen, und jede dieser folgenden Zeilen enthält einen String (maximal 255 Buchstaben). Der RMaker nimmt die Angabe für die Anzahl der folgenden Strings sehr genau: wenn sie größer ist als die tatsächliche Anzahl der folgenden Zeilen, werden wahrscheinlich Zeilen aus den Beschreibungen der folgenden Resourcen verschluckt (und es gibt einen Folge-Fehler, da deren Format nicht mehr stimmt).

```
TYPE MENU
ResName ,ResID (ResAttr)
MenuTitel
     MenuPunkt1
     MenuPunkt2
     .....
     MenuPunktN
```

Resourcen vom Typ 'MENU' enthalten die komplette Datenstruktur, wie sie auch am Ende eines MenuHandles zu finden ist. In ihr werden der Titel, die Texte für die einzelnen Menü-Punkte, deren Eigenschaften usw. abgelegt. Die entsprechende Resource-Beschreibung enthält in der ersten Zeile den Text des Menü-Titels und in den folgenden Zeilen die Angaben zu den einzelnen Punkten des Menüs. Das Ende der Spezifikation eines Menüs wird durch eine Leerzeile markiert.

Die Zeilen, die die Menü-Punkte definieren, können das gleiche Aussehen haben wie die Texte, die an die Prozedur **AppendMenu** übergeben werden (der Resource-Compiler verwendet auch die Prozedur **AppendMenu** zum Aufbau der Resourcen). Sie können auch Sonderzeichen wie '(','<' und '/' enthalten, die das Aussehen der Punkte modifizieren bzw. andere Änderungen an diesen Punkten durchführen. (Vgl. hierzu das Kapitel *Menüs*.)

```
TYPE WIND
ResName ,ResID (ResAttr)
FensterTitel
oben links unten rechts
{ Visible , Invisible } { GoAway , NoGoAway }
FensterTyp
RefCon
```

Resourcen vom Typ 'WIND' enthalten eine Datenstruktur, die alle nötigen Angaben zur Erzeugung eines Fensters enthält. Es handelt sich dabei um eine kompakte Speicherung der Parameter von **NewWindow**. Resourcen dieses Typs werden nicht wirklich zu Fenstern (WindowRecords), sondern enthalten nur die Angaben für das Aussehen des Fensters. Es ist deshalb auch sinnvoll, sie über eine Angabe der Resource-Attribute **purgeable** zu machen, da sie nach der Erzeugung des wirklichen Fensters nicht mehr benötigt werden.

Die erste Zeile der Resource-Beschreibung enthält den Namen des Fensters. Dieser ist natürlich nur dann sichtbar, wenn das Fenster eine Titel-Leiste hat, in der der Name gezeigt wird. Die nächste Zeile enthält das Rechteck, in globalen Koordinaten, das das Fensterinnere auf dem Bildschirm einnehmen soll. Man achte bei der Angabe des Rechtecks darauf, daß die übliche Reihenfolge für Koordinatenangaben aus technischen Gründen vertauscht ist. Für jeden Punkt wird zunächst die vertikale und dann die horizontale Koordinate angegeben – genau umgekehrt wie bei den Prozeduren von QuickDraw oder des WindowManagers.

Die dritte Zeile enthält die Angaben, ob das Fenster sichtbar (**Visible**) oder unsichtbar (**InVisible**) ist und ob es ein Schließkästchen erhält (**GoAway**) oder nicht (**NoGoAway**). Sichtbar machen kann man ein Fenster mit dem entsprechenden Prozedur-Aufruf schnell, die Angabe für das Schließkästchen ist hingegen permanent.

Die vierte Zeile enthält die Nummer des Fenster-Typs, der das Aussehen des Fensters bestimmt, und die fünfte Zeile den Wert für das Feld **refCon** des WindowRecords. Die Bedeutung dieser Parameter kann im Kapitel *Fenster-Verwaltung* nachgesehen werden.

Die folgenden Resource-Typen können mit dem RMaker nicht oder nur sehr umständlich erzeugt werden. Sie werden aber auch relativ häufig benötigt, weswegen sie hier aufgelistet werden. Eine Änderung von Resourcen dieser Typen ist zumeist mit dem Resource-Editor *ResEdit* relativ problemlos in interaktiver Form möglich.

```
TYPE PICT
ResName ,ResID (ResAttr)
????
```

Eine Resource vom Typ 'PICT' enthält ein gespeichertes QuickDraw-Picture. Resourcen dieses Typs werden im allgemeinen nicht mit dem Resource-Editor erzeugt, da das Format solcher Pictures nicht dokumentiert ist (s.u.).

```
TYPE PAT
ResName ,ResID (ResAttr)
????
```

Eine Resource vom Typ 'PAT ' enthält ein QuickDraw-Muster. Ein Muster (Pattern) ist ein sehr simples BitImage der Größe 8 * 8 Punkte. Die Resource-Daten bestehen dementsprechend aus insgesamt 8 Bytes.

```
TYPE PAT#
ResName ,ResID (ResAttr)
AnzahlMuster
????
```

Eine Resource vom Typ 'PAT#' enthält eine Liste von QuickDraw-Mustern. Die ersten zwei Bytes der Resource-Daten enthalten die Anzahl der folgenden Muster in Form einer 16-Bit-Zahl, danach folgen diese selbst (jeweils 8 Byte).

```
TYPE ICON
ResName ,ResID (ResAttr)
????
```

Eine Resource vom Typ 'ICON' enthält das BitImage eines Icons. Dies ist ein BitImage der Größe 32 x 32 Pixel, die Resource-Daten haben deshalb eine Größe von 128 Byte.

```
TYPE CURS
ResName ,ResID (ResAttr)
????
```

Eine Resource vom Typ 'CURS' enthält die Beschreibung eines Maus-Cursors. Darin wird festgelegt, wie der Cursor genau aussieht, wenn er weiße bzw. schwarze Flächen auf dem Bildschirm überdeckt, und welcher Punkt des 16 * 16 Rechtecks, das den Cursor umschließt, mit den Maus-Bewegungen synchronisiert wird.

9.1.6 Die Operationen des ResourceManagers

Es fogt nun eine Liste der wichtigsten Operationen des ResourceManagers. Wie üblich, werden die seltener benötigten oder "gefährlicheren" Operationen fortgelassen.

Wie schon weiter oben erläutert wurde, besteht die Kennung eines Resource-Typs aus vier Buchstaben. In Pascal sieht die Definition dafür aus wie folgt:

```
TYPE ResType= PACKED ARRAY [1..4] OF CHAR;
```

Man kann allerdings nicht in jeder Programmiersprache davon ausgehen, daß es möglich ist, vier Buchstaben in solch kompakter Form (es müssen genau 32 Bit sein) abzuspeichern und sie trotzdem so einfach wie in LisaPascal angeben zu können (z.B. 'MENU'). Dazu aber mehr im Anhang dieses Buches.

```
FUNCTION ResError: INTEGER;
```

Der ResourceManager versucht, jede Operation, wie vom aufrufenden Programm gewünscht, durchzuführen. Gelingt ihm dies nicht, kann er z.B. eine gewünschte Resource nicht finden, so kann das Programm die Gründe dafür über die Funktion **ResError** erfahren. (Von Assembler aus kann der entsprechende Code direkt aus der globalen Variablen **ResError** gelesen werden.) **ResError** kann einige ResourceManager-spezifische Codes liefern, aber auch Codes, die zu anderen Paketen der ToolBox gehören (und dort nachgeschlagen werden müssen). Passen die Daten einer benötigten Resource z.B. nicht mehr in den Heap, so ergibt **ResError** nach dem entsprechenden Aufruf **memFullErr**. Die resourcespezifischen Codes sind die folgenden:

```
CONST
    noErr           = 0;    {Alles OK}
    ResNotFound     = -192; {Resource   ex. nicht}
    ResFNotFound    = -193; {Res.-Datei ex. nicht}
    AddResFailed    = -194;
        {Operation AddResource    ist mißlungen}
    RmvResFailed    = -196;
        {Operation RemoveResource ist mißlungen}
    DskFullErr      = -34;
            {Resource paßt nicht mehr die Diskette}
```

```
FUNCTION  OpenResFile
            (fileName:  Str255): INTEGER;

PROCEDURE CloseResFile(refNum:  INTEGER);
```

Diese beiden Funktionen erlauben das Öffnen und Schließen zusätzlicher Resource-Dateien. **OpenResFile** eröffnet eine zusätzliche Resource-Datei (bzw. die Resource-Gabel der Datei **fileName**) und liefert eine Nummer zurück, die diese Resource-Gabel eindeutig kennzeichnet (**refNum**), falls das Öffnen gelingt. Diese Nummer kann verwandt werden, um die Datei nach Gebrauch wieder zu schließen.

Ein Öffnen zusätzlicher Resource-Dateien ist nur sehr selten (z.B. während der Programm-Entwicklung) nötig. Alle Resourcen, die ein Programm benötigt, sollten normalerweise in der Programm-Datei selbst oder in der Resource-Gabel der Datei "System" liegen. Diese beiden Dateien sind sowieso schon beim Start des Programms geöffnet.

CloseResFile wird noch seltener gebraucht. Diese Operation schließt die Resource-Gabel mit der angegebenen Nummer (**refNum**). Was man unbedingt vermeiden sollte (wenn man nicht genau weiß, was man tut), ist das Schließen der System-Datei, indem man für **refNum** 0 übergibt! (0 kennzeichnet bei allen Operationen, die Datei-Nummern verlangen, immer die Resource-Gabel der Datei "System".)

```
FUNCTION GetResource
            (rsrcType:  ResType;
             rsrcID:    INTEGER): Handle;

FUNCTION GetNamedResource
            (rsrcType:  ResType;
             rsrcName:  Str255): Handle;
```

Diese beiden Funktionen liefern einen Handle auf die gewünschte Resource zurück, deren Datenblock der ResourceManager dazu bei Bedarf von der Diskette in den Speicher einliest. **GetResource** identifiziert die Resource über ihren Typ und ihren Schlüssel und **GetNamedResource** verwendet statt des Schlüssels den Resource-Namen. Existiert die so spezifizierte Resource nicht in einer der im Moment offenen Resource-Dateien, liefern beide Funktionen NIL zurück.

```
PROCEDURE SetResLoad(loadRes: BOOLEAN);
```

SetResLoad steuert, ob durch **GetResource** bzw. **GetNamedResource** angeforderte Resourcen komplett in den Hauptspeicher gelesen werden oder ob zunächst nur ein Handle mit einem leeren Datenblock zurückgegeben wird. Nur wenn **loadRes** TRUE ist, werden bei den nachfolgen Aufrufen auch die Datenblöcke eingelesen. Ansonsten muß man durch einen expliziten Aufruf von **LoadResource** (s.u.) dafür sorgen, daß der leere Handle der Resource gefüllt wird.

Es kann manchmal ganz praktisch sein, einen leeren Handle auf eine Resource zu erhalten, um zunächst ihre Eigenschaften festzustellen (z.B., ob sie überhaupt noch in den Speicher paßt), bevor man sie durch **LoadResource** dann wirklich einliest.

Falls man jemals `SetResLoad(FALSE)` aufruft, sollte man dies so bald als möglich rückgängig machen, weil sich alle anderen Teile der ToolBox darauf verlassen, daß Resourcen sofort eingelesen werden.

```
PROCEDURE LoadResource(resource: Handle);
```

LoadResource sorgt dafür, daß sich der Datenblock von **resource** im Speicher befindet. Zeigt **resource** vor dem Prozedur-Aufruf bereits auf die eingelesenen Daten, geschieht gar nichts. Ist der entsprechende Datenblock

jedoch gepurgt (s.o.) oder gar nicht erst eingelesen worden, so wird er nun eingelesen. Verwendet man Resourcen, die purgeable sind, sollte man jedesmal, bevor man diese Resource verwendet, **LoadResource** mit dem Handle aufrufen, den man durch das erste **GetResource** oder **GetNamedResource** erhalten hat.

Dieser "Trick" funktioniert nur deshalb, weil beim purgen von Handles nur der Datenblock im Heap freigegeben wird, nicht aber der MasterPointer des Handles. Der Handle zeigt nach wie vor auf diesen MasterPointer (der allerdings leer ist), nachdem der entsprechende Datenblock gelöscht wurde. Der ResourceManager kann anhand des MasterPointers feststellen, auf welche Resource-Daten der MasterPointer vor dem Purgen zeigte, und diese nun wieder einlesen.

```
PROCEDURE GetResInfo
            (resource: Handle;
             VAR rsrcID:      INTEGER;
             VAR rsrcType:    ResType;
             VAR rsrcName:    Str255);

PROCEDURE SetResInfo
            (resource: Handle;
             rsrcID:      INTEGER;
             rsrcName:    Str255);
```

GetResInfo kann die Eigenschaften einer Resource feststellen, wenn man einen Handle darauf hat. **SetResInfo** kann die Eigenschaften in entsprechender Weise ändern. Man achte darauf, daß **SetResInfo** den Typ einer Resource nicht ändern kann. Ist dieser einmal festgelegt, gilt er, solange sich die Resource in der Resource-Datei befindet. **SetResInfo** kann auch keine Resource ändern, deren **resProtected**-Flag gesetzt ist.

```
FUNCTION  GetResAttrs
            (resource: Handle):      INTEGER;
PROCEDURE SetResAttrs
            (resource: Handle;
             newAttrs: INTEGER);
```

GetResAttrs stellt die Eigenschaften (Flags) einer Resource fest, und SetResAttrs erlaubt ihre Änderung. Die einzelnen Flags sind als einzelne Bits

in Form eines 16–Bit-INTEGERS codiert. Die folgenden Konstanten erlauben es, diese Bits (mit der Funktion **BitAnd**) abzuprüfen bzw. durch Addition zu einem INTEGER zusammenzusetzen.

```
CONST
    resChanged      =2;
    resPreload      =4;
    resProtected    =8;
    resLocked       =16;
    resPurgeable    =32;
    resSysHeap      =64;
```

GetResAttrs kann vor allem dann sehr sinnvoll sein, wenn man feststellen möchte, ob eine Resource **purgeable** ist.

```
FUNCTION SizeResource
             (resource: Handle):     LONGINT;
```

SizeResource stellt die Größe des Datenblocks der entsprechenden Resource fest, egal, ob sich dieser im Moment im Speicher befindet oder nicht. **SizeResource** darf nicht mit **GetHandleSize** verwechselt werden, die ja für gepurgte Datenblöcke 0 ergeben würde.

```
PROCEDURE ChangedResource(resource: Handle);
```

ChangedResource markiert den Datenblock einer Resource als geändert. Bevor diese Resource gepurgt wird oder die Resource-Gabel, die sie enthält, geschlossen wird, wird diese Änderung auf der Diskette gesichert. Wann immer man den Datenblock einer Resource modifiziert und möchte, daß diese Modifikation dauerhaft ist, sollte man sofort nach der Änderung **ChangedResource** aufrufen.

```
PROCEDURE AddResource
                  (theData: Handle;
                   theType: ResType;
                   theID:   INTEGER;
                   theName: Str255);
```

```
PROCEDURE RmveResource(resource: Handle);
```

AddResource und **RmveResource** werden benötigt, um den Inhalt von Resource-Dateien ändern zu können. **AddResource** fügt den Datenblock an **theData** als neue Resource vom Typ theType mit dem Schlüssel **theID** und dem Namen **theName** in die zuletzt geöffnete Resource-Datei ein. Der betroffene Resource-Handle wird automatisch als geändert (**resChanged**) markiert.

RmveResource entfernt den Datenblock, auf den **resource** verweist, aus der zuletzt geöffneten Resource-Datei. Ein Fehler tritt auf, falls **resource** nicht in dieser liegt.

```
PROCEDURE UpdateResFile(refNum: INTEGER);
```

UpdateResFile sichert alle Änderungen an Resourcen, der durch **refNum** identifizierten Datei auf Diskette. Diese umfaßt alle Änderungen an den Resource-Daten und zusätzlichen Merkmalen (Schlüssel, Name und Flags). **UpdateResFile** hat nur eine Wirkung, wenn mindestens eine Resource aus dieser Datei mit **ChangedResource** als geändert markiert wurde. Dann werden allerdings alle Änderungen an Resourcen (auch die an Resourcen, die nicht mit **ChangedResource** markiert wurden) gesichert. Dieselben Aktionen werden übrigens auch immer dann aufgerufen, wenn **CloseResFile** auf einer Resource-Datei aufgerufen wird – egal, ob dies explizit oder implizit (bei der Beendigung eines Programms) geschieht.

Hat man mehrere offene Resource-Dateien und möchte nur die Änderungen an einer sichern, so kann man die Nummer der Resource-Datei, die eine bestimmte Resource enthält, mit der folgenden Prozedur feststellen:

```
FUNCTION   HomeResFile
              (rsrc: Handle):refNum: INTEGER;
```

HomeResFile gibt die Nummer zurück, die die Resource-Datei, die die Resource **rsrc** enthält, gegenüber dem ResourceManager eindeutig identifiziert.

Um sicherzugehen, daß Blocks vor dem Purgen automatisch wieder im Resource-File gesichert werden (sofern sie als geändert markiert sind), sollte man die folgende Prozedur aufrufen:

```
PROCEDURE SetResPurge(save: BOOLEAN);
```

Ist **save** TRUE, wird vor dem Purgen eines Blocks geprüft, ob es sich um einen geänderten Resource-Handle handelt. Ist **save** FALSE, nicht.

Die oben beschriebenen Operationen des ResourceManagers sind diejenigen, mit denen ein Programm am häufigsten zu tun hat. Noch häufiger allerdings tritt der ResourceManager in Aktion, wenn Operationen aus anderen Paketen aufgerufen werden. So können z.B. die Beschreibungen von Fenstern und Menüs komplett in Resourcen abgelegt werden. Sie werden dann über den WindowManager bzw. MenuManager eingelesen, sobald das Programm z.B. ein neues Fenster erzeugt. Auch für einige häufig benötigte Resource-Typen sind Funktionen vorbereitet, die das Einlesen von Resourcen dieses Typs etwas vereinfachen. Sie gehören laut *Inside Macintosh* nicht mehr zum ResourceManager, sondern zu anderen Paketen der ToolBox, ihre Beschreibungen folgen aber trotzdem an dieser Stelle, da sie logisch zur ToolBox gehören. Sie liegen teilweise allerdings nicht in der ToolBox, sondern sind innerhalb der verwendeten Programmiersprache implementiert. Es kann deshalb nicht garantiert werden, daß sie in jeder Implementierung jeder Programmiersprache zur Verfügung stehen.

```
FUNCTION  GetString(id: INTEGER): StringHandle;

PROCEDURE SetString
             (str:      StringHandle;
              newStr:   Str255);
```

GetString liefert den Handle der Resource vom Typ 'STR ' mit der Nummer **id** zurück. Ein Aufruf von **GetString** entspricht im wesentlichen dem Aufruf **GetResource('STR ',id)**. **SetString** erlaubt es in bequemer Weise, einen solchen String am Handle zu ändern (und eventuell durch

ChangedResource auch in der Resource-Datei). Ein solcher *StringHandle* kann z.B. sofort für die Prozedur **DrawString** verwendet werden: **DrawString(GetString(286)^^)**.

```
PROCEDURE GetIndString
                (VAR theStr: Str255;
                 id:         INTEGER;
                 index:      INTEGER);
```

GetIndString liest die Resource vom Typ 'STR#' (String-Liste) mit der Nummer **id** ein, sucht aus dem entsprechenden Datenblock den String an der **index**-ten Stelle heraus und liefert ihn in **str** zurück.

```
FUNCTION GetPicture(id: INTEGER): PicHandle;
```

GetString liefert den Handle der Resource vom Typ 'PICT' mit der Nummer **id** zurück. Dieser Handle kann sofort mit der QuickDraw-Operation **DrawPicture** gezeichnet werden. Ein Aufruf von **GetString** entspricht im wesentlichen dem Aufruf **GetResource('PICT',id)**.

```
FUNCTION  GetPattern(id: INTEGER): PatHandle;
```

```
PROCEDURE GetIndPattern
            (VAR pat: Pattern;
             id:      INTEGER;
             index:   INTEGER);
```

GetPattern liefert den Handle der Resource vom Typ 'PAT ' (QuickDraw-Muster) mit der Nummer **id** zurück. Dieser Handle kann, nachdem er dereferenziert wurde, für die QuickDraw-Operation **PenPat** verwendet werden: **PenPat(GetPattern(......)^^)**. **GetIndPattern** entnimmt ein Muster aus der Resource **id** vom Typ 'PAT#' (Muster-Liste) und liefert es im VAR-Parameter **pat** zurück. Mindestens eine solche Muster-Liste steht jedem Programm zur Verfügung. Diese Liste hat den Schlüssel (**id**) 0, ist in der Datei "System" gespeichert und enthält die 38 Muster, die z.B. auch in MacPaint zur Verfügung stehen.

```
FUNCTION   GetIcon(id: INTEGER): IconHandle;

PROCEDURE PlotIcon(inRect: Rect; icon: IconHandle);
```

GetIcon liefert den Handle der Resource vom Typ 'ICON' (ein spezielles QuickDraw-BitImage) mit der Nummer **id** zurück. Dieser Handle kann für die Operation **PlotIcon** verwendet werden: **PlotIcon(r,GetIcon(......))**.

Die Bedeutung der Parameter von **PlotIcon** dürften klar sein. Der Typ **Icon** bzw. **IconHandle** ist nirgends in der ToolBox definiert, es handelt sich jedoch um ein BitImage (vgl. das Kapitel *QuickDraw*) mit einer Seitenlänge von 32 * 32 Pixeln (benötigt also 128 Byte). Mißt das Reckteck **r** nicht 32 * 32 Punkte, so wird diese BitMap durch **PlotIcon** in verzerrter Form gezeichnet.

Während QuickDraw-Pictures gelegentlich auch im selben Programm erzeugt werden, in dem sie dann auch gezeichnet werden, werden Icons üblicherweise immer mit einem Resource-Editor erzeugt, in die Programm-Datei gelegt und dann von diesem Programm mit **GetIcon** und **PlotIcon** verwendet.

9.2 Die Verwendung von Resourcen im Rahmenprogramm

Die Änderungen, die die Nutzung des Resource-Konzeptes am Rahmenprogramm verursachen, sind relativ gering. Sie betreffen praktisch nur die Initialisierungsphase. Anstatt hier die Angaben, die zur Bildung der Menüs und der beiden Fenster benötigt werden, direkt im Programm zu machen, werden sie aus der Resource-Gabel der Programm-Datei eingelesen.

Obwohl es sich nur um kleine Änderungen handelt, sollte ihre Bedeutung nicht unterschätzt werden. Wie an ihnen hoffentlich bereits zu sehen ist, werden einige Aspekte des Programms dadurch wesentlich simpler. So ist es z.B. nicht gerade sehr bequem, ein Menü im Programm zu bilden.

Zusätzlich wird Platz im Speicher gespart. Während in der früheren Version des Rahmen-Programms z.B. die Menü-Texte zweimal im Speicher standen — einmal im Programm-Code und einmal am entsprechenden MenuHandle — sind sie jetzt nur einmal vorhanden; nämlich im MenuHandle, der direkt aus einer Resource gewonnen wird.

Zudem wird das Programm flexibler. Es ist in der neuen Version problemlos möglich, z.B. das Aussehen eines oder beider der gezeigten Fenster zu modifizieren ohne das Programm zu ändern. Mit einem Resource-Editor kann sowohl die anfängliche Position und Größe der beiden Fenster manipuliert werden wie auch z.B. der Fenster-Typ.

Die Erörterung des ResourceManagers in diesem Kapitel war außerdem auch deshalb so ausführlich, weil er in jedem wirklichen Anwendungs-Programm, dessen Nutzen wesentlich über das vorgestellte Rahmenprogramm hinausgeht, eine sehr große Rolle spielt. Auch werden die hier vorgestellten Möglichkeiten in den nächsten beiden Kapiteln noch eine große Rolle spielen.

9.2.1 Überblick über die Änderungen

Das Programm **Rahmen2** weist nur an zwei Stellen Unterschiede gegenüber dem aus dem letzten Kapitel bekannten **Rahmen1** auf. In der Prozedur **InitProg** werden einige Angaben für Menüs und Fenster nun aus der Resource-Gabel der Programm-Datei eingelesen und in **CreatePict** wird der, in dem bewegten Bild gezeigte, Text nun ebenfalls aus einer 'STR '-Resource eingelesen.

9.2.2 Änderungen an der Prozedur InitProg

Wesentliche Änderungen an InitProg bestehen eigentlich nur aus dem Austauschen der Prozeduren **NewMenu** und **NewWindow** gegen **GetMenu** bzw. **GetNewWindow**. Die ersten beiden Prozeduren verlangen die Angabe einer kompletten Parameterliste, während die beiden anderen Prozeduren sich den größten Teil der nötigen Parameter aus der Resource-Gabel eines Programms holen.

```
CONST
    firstMenu          = 1001;
    lastMenu           = 1002;

    {geänderte Konstanten für Menü-IDs}
    mAblage            = 1001;
        { Punkte bleiben gleich }
    mBearbeiten        = 1002;
        { Punkte bleiben gleich }
```

```
    {neue Konstanten für 'WIND'-Resource-IDs}
    idWin1              = 1001;
    idWin2              = 1002;

VAR
    menu:       ARRAY[firstMenu..lastMenu]
                    OF MenuHandle;
    fenster1:   WindowPtr;
    fenster2:   WindowPtr;
    wPict:      PicHandle;
    scale:      INTEGER;


FUNCTION  CreatePict: PicHandle;
BEGIN
     ...... { Wird weiter unten beschrieben }
END;{CreatePict}

PROCEDURE InitProg;
VAR
    i:            INTEGER;
BEGIN
    MoreMasters;
    MoreMasters;
    MoreMasters;
    MoreMasters;

    FOR i:= firstMenu TO lastmenu DO
        menu[i] := GetMenu(i);
    FOR i:= firstMenu TO lastmenu DO
        InsertMenu(menu[i],0);
    DrawMenuBar;
```

```
        fenster1 := GetNewWindow
                        (idWin1, { Resource-ID }
                         NIL,
                         Ptr(-1));
        SetWKind(fenster1,grafKind);

        fenster2 := GetNewWindow
                        (idWin2, { Resource-ID }
                         NIL,
                         Ptr(-1));
        SetWKind(fenster2,grafKind);

        SetPort(fenster1);
        SelectWindow(fenster1);

        wPict := CreatePict;
        scale := 8;

    END;{InitProg}
```

Da die beiden Menüs nun in der Resource-Gabel liegen, dürfen sie keine Nummern mehr besitzen, die mit denen in Konflikt kommen, die für das System reserviert sind. Die Menü-Nummern sind nämlich gleich den Resource-Nummern ihrer Beschreibungen. Deswegen werden die entsprechenden Konstanten des Programms geändert. Die MenuHandles können nun auch in einer FOR-Schleife eingelesen werden, da der Inhalt der Menüs im Programm nicht mehr bekannt sein muß.

Auch die Erzeugung der beiden Fenster wird wesentlich simpler, da auch hier die entsprechenden Resourcen alle wichtigen Angaben beinhalten.

9.2.3 Änderungen an der Prozedur CreatePict

Um die Verwendung der sehr oft benötigten String-Resourcen zu demonstrieren, wird der Text, der innerhalb des bewegten Bildes erscheint, in einer solchen abgelegt. Die Handhabung ist denkbar einfach: Statt der Prozedur DrawString den gewünschten Text direkt zu übergeben, ruft man die Funktion GetString auf. Diese muß allerdings noch doppelt dereferenziert werden, da es sich bei ihrem Ergebnis, wie bei den meisten Resource-Funktionen, um einen Handle handelt.

```
CONST
    {Resource-ID für 'STR '-Resource}
    idStr           = 1001;

FUNCTION  CreatePict: PicHandle;
VAR
    pict:           PicHandle;
    saveClip:       RgnHandle;
    r:              Rect;
BEGIN
    SetRect(r,1,1,50,50);
    saveClip := NewRgn;
    GetClip(saveClip);
    ClipRect(r);
    pict := OpenPicture(r);
        ......
        { Anfang bleibt gleich }

        MoveTo(5,45);
        DrawString(GetString(idStr)^^);
    ClosePicture;
    CreatePict := pict;
    SetClip(saveClip);
    DisposeRgn(saveClip);
END;{CreatePict}
```

9.2.4 Listing der Resource-Gabel des Programms

Hier folgt nun das Listing einer Text-Datei, die für die Erzeugung der von **Rahmen2** benötigten Resourcen dienen kann. Das Format der Datei ist das des Programms *RMaker*, das bei den meisten der auf dem Macintosh laufenden, Programm-Entwicklungs-Systeme mitgeliefert wird. Eine vergleichbare Menge von Resourcen kann aber auch mühelos z.B. mit dem Resource-Editor *ResEdit* erzeugt werden.

Wie der Resource-Editor oder RMaker bei einer bestimmten Programmiersprache aber eingesetzt wird, ist teilweise sehr unterschiedlich. Der Leser sollte sich deshalb, bevor er das Programm **Rahmen2** wirklich austestet, ausführlich mit dem entsprechenden Teil der Dokumentation seiner Programmiersprache beschäftigen.

```
!Rahmen2
APPLRAHM

TYPE MENU
    ,1001
Ablage
    (Öffne Fenster 1/1
    Öffne Fenster 2/2
    (-
    Beenden

    ,1002
Bearbeiten
    Größer/G
    Kleiner/G
    (-
    1 nach oben
    (2 nach oben


TYPE WIND
    ,1001
Fenster 1
100 100 300 300
Visible GoAway
0
0

    ,1002
Fenster 2
110 110 310 310
InVisible GoAway
0
0
```

```
TYPE STR
    ,1001
'STR '-Resource
```

Die erste Zeile der Datei gibt an, daß die erzeugten Resourcen an die Datei **Rahmen1** angehängt werden sollen. Würde das Ausrufezeichen ('!') fehlen, würden die bereits in der Datei befindlichen Resourcen durch die neuerzeugten ersetzt werden. Dies ist nicht unsere Absicht, da wir von der Annahme ausgehen, daß sich in dieser Datei bereits der Programm-Code befindet (in Resourcen von Typ 'CODE').

Die zweite Zeile bestimmt, daß die Datei den *Datei-Typ* 'APPL' und den *Datei-Erzeuger* 'RAHM' haben soll. Der Typ 'APPL' sorgt dafür, daß der Finder unsere Programm-Datei als ausführbares Programm erkennt. Der Datei-Typ und die Kennung sind ansonsten zwei Konzepte, die im Rahmen dieses Buches nicht weiter behandelt werden.

Danach folgen Gruppen von Resourcen gleichen Typs, die der RMaker erzeugen soll; zunächst die Menüs, dann die Fenster, dann der einzelne String. Jede Gruppe beginnt mit der Angabe "**TYPE** XXX", und in den darauf folgenden Zeilen kommen dann die Beschreibungen der Resourcen dieses Typs.

Das Format dieser Beschreibungen ist in einem vorangegangenen Abschnitt bereits ausführlich erläutert worden und soll hier nicht wiederholt werden. Jede Resource beginnt mit der Angabe ihres Schlüssel (davor optional ein Name) und hört im allgemeinen mit einer Leerzeile auf.

Interessant sind allerdings die Angaben für die MENU-Resourcen, die bereits das initiale Aussehen der Menü-Punkte bestimmen. Insbesondere werden Menü-Punkte, die bereits beim Start des Programms nicht aufgerufen werden können, durch Angabe einer Klammer '(' grau gezeichnet.

9.3 Schlußbemerkung

Dieses Kapitel hat hoffentlich gezeigt, wie vielseitig das Resource-Konzept des Macintosh ist, obwohl nur ein kleiner Teil der Verwendungs - möglichkeiten angeschnitten werden konnte. Prinzipiell sollte jedes Programm soviel seiner Daten wie möglich in Resourcen ablegen. Wenn diese Daten für Darstellungen verwendet werden, die der Benutzer zu sehen bekommt, ist dies sogar äußerst wichtig, da es ein Anpassen des Programms an geänderte Gegebenheiten (z.B. eine andere Muttersprache des Benutzers) meist viel leichter macht.

Hierzu können nicht nur die standardmäßig vordefinierten Resource-Typen verwendet werden, sondern auch neue jederzeit definiert werden. Mindestens einer der jetzt schon vorhandenen Resource-Editoren läßt sich problemlos auch an neue Resource-Typen anpassen.

Resourcen können auch dazu verwendet werden, einen Programmzustand abzuspeichern. Gewisse Einstellungen eines Programmes können hierfür in Resourcen abgelegt werden und bleiben nach dem Verlassen des Programms erhalten. Startet der Benutzer das Programm das nächste Mal, gelten die bei der letzten Benutzung gewählten Einstellungen nach wie vor. Solche Einstellungen (Resourcen) können sowohl bei Dokumenten (z.B. Papierformat) wie auch beim Programm selbst liegen.

Der Leser sollte vielleicht, bevor er im Buch fortfährt, noch ein wenig mit den Möglichkeiten für Resourcen experimentieren – "um ein Gefühl dafür zu bekommen". Eine Möglichkeit, die sich geradezu aufdrängt, wäre es z.B., das Bild, das mit der Maus bewegt werden kann, aus einer Resource einzulesen. Es könnte z.B. mit MacPaint oder MacDraw gezeichnet werden und wesentlich effektvoller aussehen als das simple Bildchen, das im Rahmenprogramm jetzt verwendet wird. (Hierzu ein kleiner Tip: Das Bild in MacDraw oder MacPaint zeichnen, ausschneiden und ins Album einsetzen. Das Album erzeugt eine Datei namens "Album Datei" bzw. "Scrapbook File". In dieser liegt das Bild als Resource vom Typ 'PICT' und kann problemlos mit dem Resource-Editor in die Programm-Datei bewegt werden.)

10 Schreibtisch-Utensilien – die dritte Ausbaustufe

Eine Einrichtung, die sich inzwischen auch auf anderen Mikrocomputern größter Beliebtheit erfreut, sind die Schreibtisch-Utensilien. Schreibtisch-Utensilien sind kleine Programme, die man durch einen möglichst simplen Handgriff jederzeit aufrufen und verwenden kann, ohne das Programm, in dem man sich zur Zeit befindet, zu verlassen. Typische Anwendungen hierfür schließen z.B. Taschenrechner, Terminkalender, Notizbücher u.ä. ein. Es sind also die vertrauten Utensilien, mit denen man als "Schreibtischtäter" tagtäglich umgeht – daher auch der Name "Schreibtisch-Utensilien" (oder auf englisch "***Desk Accessories***" oder kurz "*DAs*" bzw. im Singular "*DA*"). Ein anderer Name für solche Hilfsmittel ist z.B. auch "PopUp-Programm", da sie jederzeit auf dem Bildschirm auftauchen und wieder verschwinden können.

Diese Mini-Programme sind einfach zu nützlich, als daß es ein Programm geben dürfte, in dem sie nicht funktionieren. Auf dem Macintosh gibt es z.B. sogar Text-Verarbeitungen, Tabellen-Kalkulations-Programme, Terminal-Emulatoren und Grafik-Pakete (und natürlich viele andere hilfreiche Utilities) in Form von Schreibtisch-Utensilien. Ein Benutzer, der auf solche Hilfsmittel angewiesen ist, wird bestimmt verärgert sein, wenn er feststellt, daß er sie in einem wichtigen Programm nicht nutzen kann.

Während Schreibtisch-Utensilien auf anderen Computern sich durch trickreiche Programmierung in andere Programme "einschleichen" müssen, sind die Vorrichtungen zur Unterstützung von Schreibtisch-Utensilien auf dem Macintosh bereits in das Betriebssystem (bzw. die ToolBox) eingebaut. Das entsprechende Paket, das die wichtigsten Aspekte von Schreibtisch-Utensilien in standardisierter Form handhabt, ist der DeskManager.

Dies hat Vor- und Nachteile gegenüber vollkommen selbständigen DAs. Während diese sich wirklich in (fast) jedes Programm einschleichen können (durch Eingriffe auf Assembler-Ebene in das Betriebssystem), müssen die Schreibtisch-Utensilien auf dem Macintosh vom Programm aus unterstützt werden. Dafür ist die Bedienung, z.B. der Aufruf auf dem Macintosh, aber

auch standardisiert, und die Gefahren, die durch Programme auftauchen können, die das Betriebssystem modifizieren, sind nicht vorhanden.

Ähnlich wie die Fenster-Verwaltung (der WindowManager) einem Programm einen Großteil der Mühe mit Fenstern abnimmt, nimmt ihm der DeskManager den größten Teil der mit DAs verbundenen Arbeit ab, wenn man nur in den richtigen Momenten die richtigen Prozeduren aufruft.

10.1 Beschreibung des DeskManagers

Der DeskManager in der ToolBox des Macintosh hat die Aufgabe, dem eigentlichen Anwendungs-Programm den Umgang mit Schreibtisch-Utensilien (DAs) zu vereinfachen. Hierzu bietet er eine Reihe von Operationen an, die von einem Programm nur im rechten Moment aufgerufen werden müssen, um das komplette Standardverhalten zu realisieren, wie es für Schreibtisch-Utensilien in *Inside Macintosh* definiert ist.

10.1.1 Eine typische Schreibtisch-Utensilie

Zur Verdeutlichung der "Bedürfnisse" einer Schreibtisch-Utensilie, die entweder direkt durch das Programm oder über den DeskManager erfüllt werden müssen, soll nun eine Schreibtisch-Utensilie, die jedem Leser bekannt sein dürfte, ausführlich vorgestellt werden.

Das Kontrollfeld ist eine Schreibtisch-Utensilie, mit der die wichtigsten Parameter für die Bedienung des Macintosh in interaktiver, grafischer Form (wie es sich für den Mac gehört) vom Benutzer verändert werden können. Hierzu gehören z.B. die Laustärke des eingebauten Lautsprechers und das Muster, mit dem nicht von Fenstern verdeckte Flächen des Bildschirms gefüllt werden sollen.

Wie jede andere Schreibtisch-Utensilie wird das Kontrollfeld aufgerufen, indem man das erste Menü in der Menü-Leiste (das üblicherweise einen kleinen Apfel als Namen hat) herunterklappt und den Punkt "Kontrollfeld" daraus aussucht.

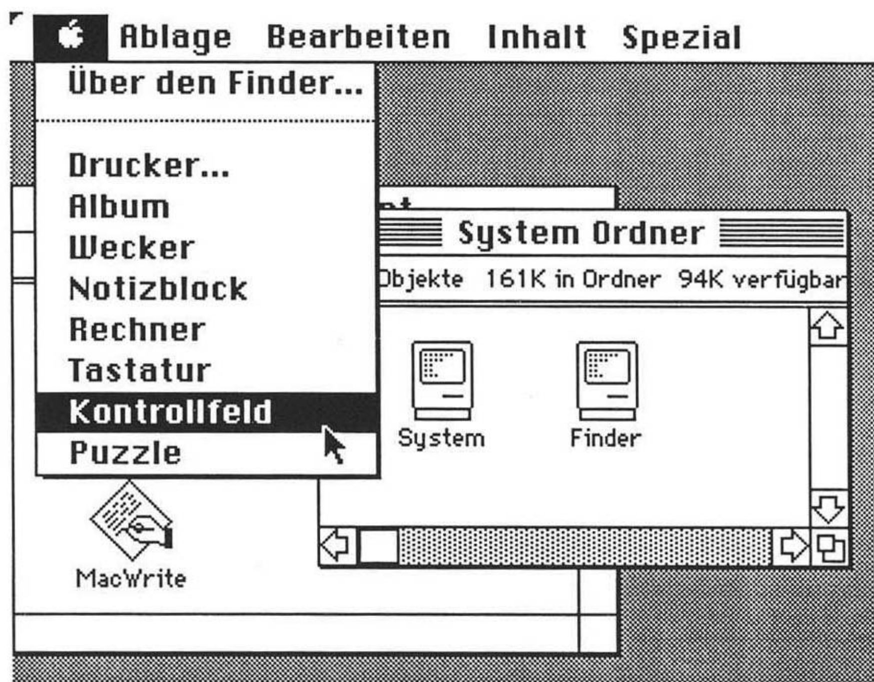


Bild 10 - 1: *Auswahl einer Schreibtisch-Utensilie aus dem "Apfel"-Menü*

Der Aufbau dieses sogenannten "Apfel"-Menüs ist in allen Programmen im wesentlichen gleich. Der erste Punkt dieses Menüs beginnt meist mit dem Wort "Über". Ruft man diesen Punkt auf, erhält man Informationen über das gerade laufende Programm (mit von Programm zu Programm stark schwankender Detailiertheit). Unter diesem Punkt kommt ein Trennstrich im Menü, und dann folgen die Namen der im Augenblick zur Verfügung stehenden Schreibtisch-Utensilien. Ruft man einen dieser Punkte auf, so wird das entsprechende Schreibtisch-Utensil geöffnet. Dies bedeutet üblicher - weise, aber nicht notwendigerweise, auch das Öffnen eines Fensters, in dem die Utensilie gezeigt wird. In diesem Fall ist es das Fenster des Kontrollfeldes.

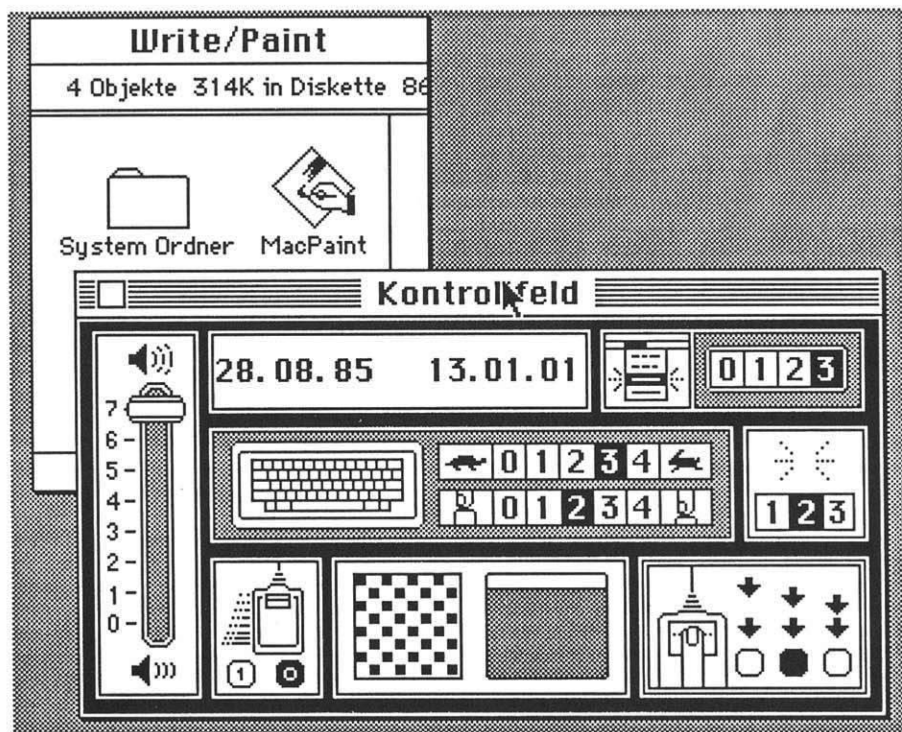


Bild 10 - 2: *Das Kontrollfeld ist geöffnet*

Dieses Fenster kann nur Informationen enthalten oder, wie beim Kontrollfeld, auch Möglichkeiten, mit Maus-Klicks oder Tastendrücken bestimmte Wirkungen zu erzielen. Die Fenster der meisten DAs lassen sich beliebig verschieben und üblicherweise auch durch einen Klick in ein Schließ-Kästchen des Fensters wieder vom Bildschirm entfernen.

Eigentlich sollte ein Programm auch erlauben, die Fenster von DAs zur Seite zu legen, d.h. ein anderes Fenster nach oben zu holen und in diesem weiterzuarbeiten, bis das DA wieder benötigt wird. DA-Fenster können deshalb verdeckt und auch wieder freigelegt werden. Einige Programme erlauben jedoch nur die wahlweise Benutzung der vom Programm erzeugten Fenster oder einer Schreibtisch-Utensilie. Sobald ein anderes Fenster nach oben geholt wird, wird das Fenster des DAs geschlossen. Das Rahmen -

programm in der dritten Ausbaustufe wird jedoch beliebige Kombinationen von Programm- und DA-Fenstern unterstützen und auch eine beliebige Anzahl von gleichzeitig aktiven DAs zulassen.

Das Kontrollfeld enthält unter anderem eine Uhren-Anzeige, die auch dafür verwendet werden kann, die System-Uhr des Macintosh zu stellen. Diese Uhr läuft weiter, auch wenn das Kontrollfeld-Fenster nicht an oberster Stelle liegt und/oder teilweise von anderen Fenstern verdeckt wird. Auch wenn die Uhrzeit sich aus anderen Gründen ändert (z.B. weil sie über eine anderes DA geändert wird), ändert sie sich entsprechend in der Anzeige des Kontrollfeldes mit.

Alle Schreibtisch-Utensilien unterstützen ferner standardmäßig das Menü ***Bearbeiten***, wie man es z.B. aus MacWrite oder MacPaint kennt. Die erkannten Befehle hieraus sind: ***Widerrufen***, ***Ausschneiden***, ***Kopieren***, ***Einsetzen*** und ***Löschen***. Nicht alle DAs erkennen oder benötigen diese Befehle. Wo sie aber sinnvoll eingesetzt werden können, z.B. bei DAs, in deren Fenstern Texte editiert werden, werden diese Befehle auch erkannt und ausgeführt. Wichtig hierbei ist allerdings auch, daß Texte und Bilder, die im Fenster eines DAs ***ausgeschnitten*** oder ***kopiert*** wurden, auch im Programm ***eingesetzt*** werden können und umgekehrt – soweit dies sinnvoll ist.

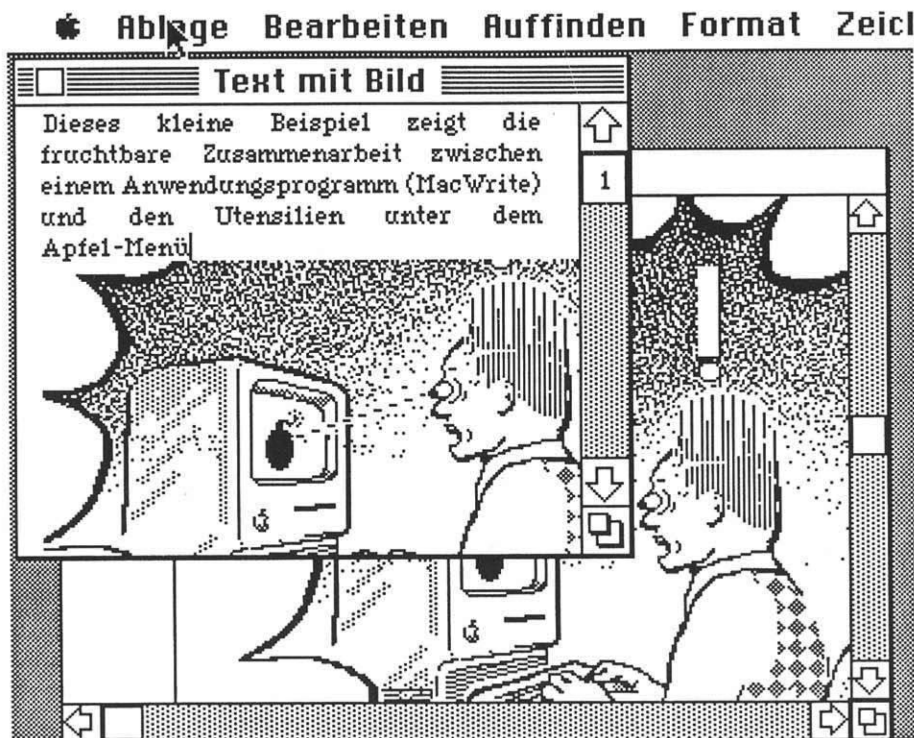


Bild 10 - 3: *Kopieren und Einsetzen zwischen DAs und Programmen*

10.1.2 Anforderungen von Schreibtisch-Utensilien

Aus dieser Beschreibung der typischen Eigenschaften einer Schreibtisch-Utensilie ergeben sich eine Reihe von Anforderungen, die diese nützlichen Mini-Programme an das eigentliche Anwendungsprogramm stellen. Zumeist handelt es sich dabei um ein Weiterreichen von bestimmten Ereignissen an die betroffenen DAs.

Die erste und wichtigste Anforderung an das Programm ist natürlich das explizite Starten des DAs. Hierzu wird nur der Name des DAs benötigt; den Rest erledigt eine Prozedur des DeskManagers. DAs sollten aber nie "einfach so" gestartet werden, sondern immer nur, wenn der Benutzer dies wünscht. Die übliche Methode, ihm die zur Auswahl stehenden Schreibtisch-Utensilien

zu zeigen, ist dabei das Apfel-Menü. Wählt er hier den Namen eines DAs aus, so muß das Programm diesen Namen feststellen (den Text des gewählten Menü-Punktes) und damit das DA öffnen. Zum Füllen eines Menüs mit den Namen aller zur Verfügung stehenden Schreibtisch-Utensilien kann eine besondere Prozedur des MenuManagers verwendet werden, die weiter unten erläutert wird.

Ansonsten müssen natürlich alle Ereignisse, die das DA betreffen (bzw. dessen Fenster), an dies weitergeleitet werden. Hierzu gehören Tastendrücke und Maus-Klicks, wenn das Fenster eines DAs oben liegt, und natürlich **updateEvts** und **activateEvts** für Fenster des DAs, die vom Hauptprogramm aus natürlich nicht bearbeitet werden können.

Dieses Weiterleiten von Ereignissen geschieht zu einem großen Teil "hinter den Kulissen" der ToolBox, ohne daß sich das Programm darum kümmern müßte. Die Funktion **GetNextEvent** spielt eine große Rolle dabei – Programme, die **GetNextEvent** nicht aufrufen, können keine Schreibtisch-Utensilien nutzen.

Maus-Klicks im Fenster eines DAs und Menü-Befehle, die sich ja auch an ein DA richten können, müssen explizit vom Programm aus durch entsprechende Aufrufe des DeskManagers an DAs weitergereicht werden. Keine Schreibtisch-Utensilie kann ja z.B. wissen, welche Nummer das Menü ***Bearbeiten*** in einem Programm hat, und doch müssen die Befehle aus diesem Menü an ein DA weitergeleitet werden, wenn dessen Fenster an oberster Stelle liegt.

Zu guter Letzt müssen natürlich auch die periodischen Anforderungen einer Schreibtisch-Utensilie erfüllt werden. Um z.B. eine laufende Uhr oder einen blinkenden Strich oder Punkt im Fenster eines DAs zu ermöglichen, muß dieses regelmäßig die Möglichkeit erhalten, "kurz anzuspringen", diese Änderungen im Fenster durchzuführen und dann wieder zum aufrufenden Programm zurückzukehren. Dies muß so oft wie nur möglich passieren, denn es kann ja DAs geben, in deren Fenstern sehr schnelle Vorgänge angezeigt werden oder die sehr schnell auf irgendwelche Änderungen reagieren müssen. Eine Faustregel hierfür lautet, daß ein Programm im allgemeinen mindestens 60mal in der Sekunde DAs die Möglichkeit einräumen sollte, zu laufen. Eine 60stel Sekunde hört sich zwar nach einem sehr kleinen Zeitabschnitt an. Dies ist für einen schnellen Prozessor, wie ihn der Macintosh besitzt, aber weitaus genug Zeit, um auch größere Aufgaben zu erledigen.

10.1.3 Die Operationen des DeskManagers

Der DeskManger stellt nur eine kleine Zahl von Operationen zur Verfügung, von denen zudem noch ein Teil üblicherweise nur von anderen ToolBox-Teilen genutzt wird, für das Rahmenprogramm deshalb also uninteressant ist. Die letzte der im folgenden beschriebenen Prozeduren gehört allerdings nicht zum DeskManager selbst, sondern zum MenuManager, der ja bereits im Kapitel *Menüs* erläutert wurde. Sie arbeitet zudem auch noch eng mit dem ResourceManager zusammen, der im letzten Kapitel beschrieben wurde.

```
FUNCTION OpenDeskAcc(theAcc: Str255): INTEGER;
```

Die Funktion **OpenDescAcc** versucht, eine Schreibtisch-Utensilie mit dem Namen **theAcc** zu starten (öffnen), und liefert, falls dies gelingt, eine Zahl zurück, die diese Utensilie eindeutig kennzeichnet. Dieser Wert wird jedoch nirgends benötigt und kann deshalb ignoriert werden. (Er ist 0, wenn das Öffnen mißlingt — aus welchen Gründen auch immer.)

OpenDeskAcc leistet alles, was vom Programm aus nötig ist, um das entsprechende DA in den Speicher zu laden und zu starten. Alles Weitere ist Aufgabe des DAs selber.

```
PROCEDURE CloseDeskAcc(refNum: INTEGER);
```

Mit dieser Prozedur kann das Mini-Programm (DA), das durch **refNum** identifiziert wird, vom Programm aus "gewaltsam" beendet werden. Die Nummer **refNum** kann, bei Schreibtisch-Utensilien, die ein Fenster öffnen, durch das Feld **windowKind** des entsprechenden WindowRecords festge - stellt werden. Deshalb ist es auch nicht nötig, sich diese beim Öffnen zu merken.

Das Schließen eines DAs durch **CloseDeskAcc** ist ein relativ seltener Spezialfall, weil sich diese meist von selbst beenden, sobald ein bestimmtes Ereignis eintritt (das natürlich vom Programm korrekt weitergeleitet werden muß). Ein solches Ereignis wäre z.B. ein Maus-Klick in das Schließkästchen eines DA-Fensters.

```
PROCEDURE SystemClick
                 (theEvent:  EventRecord;
                  theWindow: WindowPtr);
```

SystemClick ist eine der wichtigsten Operationen, mit denen Events an DAs weitergereicht werden. Wann immer ein Maus-Klick in das Fenster eines DAs fällt – die **FindWindow**-Funktion des WindowManagers gibt dann den Code **inSysWindow** zurück — sollte ein Programm **SystemClick** mit dem entsprechenden EventRecord und Fenster aufrufen. **InSysWindow** bedeutet immer einen Klick in das Fenster eines DAs, andere System-Fenster gibt es (noch?) nicht.

SystemClick bearbeitet alle Operationen mit Fenstern von DAs automatisch. Sie bewegt Fenster, holt hintenliegende Fenster nach vorn und schließt sie, wenn der Klick in das Schließkästchen fiel. War das DA-Fenster schon das oberste und der Klick fiel in das Fenster-Innere, reicht **SystemClick** das Ereignis an den Code des DAs weiter. In allen anderen Fällen findet die Behandlung innerhalb der ToolBox statt, unabhängig von dem DA, um das es sich dreht.

```
FUNCTION SystemEdit(editCmd:  INTEGER):BOOLEAN;
```

SystemEdit sollte immer dann aufgerufen werden, wenn einer der ersten fünf Befehle aus dem Menü ***Bearbeiten*** aufgerufen wurde; ***Widerrufen***, ***Ausschneiden***, ***Kopieren***, ***Einsetzen*** oder ***Löschen***. Für die verschiedenen Befehle sollten die folgenden Werte für **editCmd** übergeben werden:

Widerrufen	0
Ausschneiden	2
Kopieren	3
Einsetzen	4
Löschen	5

Sind die Befehle so angeordnet, wie z.B. im ***Bearbeiten***-Menü des Finders — Widerrufen als erster Punkt, dann ein Trennstrich und darauf die vier anderen Befehle — so kann man einfach die Nummer des Punktes minus 1 übergeben.

SystemEdit entscheidet, ob der Befehl an eine Schreibtisch-Utensilie gehen sollte, und liefert als Ergebnis TRUE, wenn das der Fall ist, und sonst

FALSE. Liefert **SystemEdit** TRUE, sollte sich das Programm nicht weiter um diesen Menü-Befehl kümmern. Liefert **SystemEdit** FALSE, sollte das Programm diesen Befehl selbst bearbeiten, sofern dies angebracht ist.

```
PROCEDURE SystemTask;
```

SystemTask ist eine Prozedur, die jedes Programm so oft wie möglich aufrufen sollte. Vor jedem Aufruf von **GetNextEvent** in der Haupt-Ereignis-Schleife z.B. ist ein **SystemTask**-Aufruf angebracht. Kann jedoch nicht garantiert werden, daß das Programm innerhalb jeder 60stel Sekunde einmal die Haupt-Ereignis-Schleife durchläuft, sollte **SystemTask** auch noch an anderen Stellen aufgerufen werden — in lang andauernden Schleifen zum Beispiel.

SystemTask gibt allen aktiven DAs (die geöffnet und noch nicht wieder geschlossen wurden) die Möglichkeit, einmal "kurz anzuspringen". Typischerweise werden dadurch Veränderungen, die sich in der Zwischenzeit ergeben haben, im Fenster des DAs widergespiegelt. (Z.B. kann der Zeiger einer Uhr weiterbewegt werden oder eine Digital-Anzeige umspringen — auch ohne, daß das DA-Fenster das oberste, aktive Fenster ist.)

Dies waren dann schon alle Prozeduren des DeskManagers, die benötigt werden, um Schreibtisch-Utensilien vom Programm aus zu unterstützen. Alles andere geschieht hinter den Kulissen. So werden andere Ereignisse als **mouseDown**-Events noch innerhalb der Prozedur **GetNextEvent** daraufhin geprüft, ob sie zu einem DA gehören, und gegebenenfalls weiter - geleitet.

Eine weitere Routine benötigt ein Programm allerdings noch, um dem Benutzer überhaupt Schreibtisch-Utensilien anbieten zu können. Diese Routine nimmt dem Programm die Hauptarbeit bei der Erzeugung des Apfel-Menüs ab.

```
P
```

```
PROCEDURE AddResMenu
                (menu:      MenuHandle;
                 theType:   ResType);
PROCEDURE InsertResMenu
                (menu:      MenuHandle;
                 theType:   ResType;
                 afterItem: INTEGER);
```

AddResMenu hängt an die Menü-Punkte, die sich schon in **menu** befinden, unten die Resource-Namen derjenigen Resourcen an, die zur Zeit von dem Resource-Typ **theType** in allen offenen Resource-Dateien zu finden sind. **Menu** muß vorher aber bereits mit **NewMenu** oder **GetNewMenu** erzeugt worden sein.

Genauso fügt **InsertResMenu** die Resource-Namen derjenigen Resourcen, die zur Zeit von dem Resource-Typ **theType** in allen offenen Resource-Dateien zu finden sind, in die Punkte eines Menüs ein. Und zwar erscheinen die neuen Menü-Punkte unter dem alten Menü-Punkt mit der Nummer **afterItem**. Die Menü-Punkte, die vorher unterhalb **afterItem** lagen, werden dabei nach unten verschoben.

Der Nutzen dieser beiden Prozeduren für die Behandlung von Schreibtisch-Utensilien liegt darin, daß es sich bei diesen um Resourcen des Typs 'DRVR' (für *Driver*) handelt. Die entsprechenden Resource-Namen sind gleichzeitig die DA-Namen, unter denen sie mit **OpenDeskAcc** geöffnet werden können. Mittels **AddResMenu** bzw. **InsertResMenu** kann deshalb auf sehr einfache Weise ein Menü mit allen gerade zur Verfügung stehenden Schreibtisch-Utensilien aufgebaut werden.

Dies muß auch deshalb auf die hier beschriebene flexible Weise geschehen, da sich die zur Verfügung stehenden Schreibtisch-Utensilien von System zu System oder gar von Programm zu Programm ändern können. Das Apfel-Menü darf deshalb nicht aus einer festen, unveränderlichen Liste von DA-Namen bestehen.

Die zweite häufigere Anwendung für diese beiden Prozeduren ist es, ein Menü mit den Namen aller gerade zur Verfügung stehenden Zeichensätze (Resourcen vom Typ 'FONT') aufzubauen. Da das Rahmenprogramm aber keine unterschiedlichen Zeichensätze oder andere Resource-Typen verwendet, bleibt das Apfel-Menü die einzige Anwendung von **AddResMenu** bzw. **InsertResMenu** darin.

10.2 Nutzung des DeskManagers im Rahmenprogramm

Nach den in den folgenden Abschnitten beschriebenen Änderungen sollte das Rahmenprogramm **Rahmen3** fähig sein, jede beliebige Schreibtisch-Utensilie zu unterstützen, die korrekt geschrieben und korrekt im System installiert wurde. Dies ist ein entscheidender Beitrag zur Nützlichkeit des Programms und sollte in jede "echte" Anwendung, die darauf aufbauend erstellt wird, übernommen werden.

10.2.1 Überblick über die Änderungen

Die Unterstützung von Schreibtisch-Utensilien verlangt eine ganze Reihe von Änderungen an den verschiedensten Stellen des Rahmen-Programms und auch in der Resource-Gabel des Programms. Glücklicherweise sind nahezu alle diese Änderungen allgemeingültig und können so, wie hier vorgestellt, in fast jedem Programm eingesetzt werden, das mit einer Haupt-Event-Schleife arbeitet.

10.2.2 Änderungen an der Resource-Datei

In der Resource-Gabel des Programms verlangt der Einsatz von Schreibtisch-Utensilien einige Änderungen an den Menüs. Am wichtigsten ist der Einbau eines Apfel-Menüs in die Menü-Liste, aus dem der Benutzer später ein DA zum Öffnen auswählen kann. Weiterhin sind Änderungen am Menü Bearbeiten nötig, damit dieses das Aussehen erhält, wie DAs es erwarten. Die alten Punkte aus diesem Menü rutschen dafür weiter nach unten.

```
TYPE MENU
     ,1000
\14
     Über das Rahmenprogramm...
     (-

     ,1001
Ablage
     (Öffne Fenster 1/1
     Öffne Fenster 2/2
     (-
     Beenden
```

```
    ,1002
Bearbeiten
    Widerrufen
    (-
    Ausschneiden
    Kopieren
    Einsetzen
    Löschen
    (-
    Größer/G
    Kleiner/G
    (-
    1 nach oben
    (2 nach oben
```

Wie üblich, wird auch hier wieder die Resource-Gabel im RMaker-Format beschrieben. Die einzigen Resourcen, die geändert werden, sind diejenigen vom Typ 'MENU'; alle anderen bleiben gleich und werden hier auch nicht noch einmal aufgelistet.

Das Apfel-Menü ist das mit der Nummer 1000. Damit wirklich ein Apfel als Name erscheint, ist in der entsprechenden Zeile '\14' eingetragen. Dies ist für den RMaker das Signal, hier das Zeichen mit der Nummer (dem ASCII-Code) 14 einzusetzen. Dies ist genau das Apfel-Symbol (das über die Tastatur nicht erzeugt werden kann). Die einzigen Menü-Punkte, die in der Resource-Beschreibung für dieses Menü definiert werden, sind der Befehl ***Über das Rahmenprogramm...*** und ein Strich. Die DA-Namen unter dem Strich werden erst beim Start des Programms angehängt. Der erste Befehl im Apfel-Menü ***(Über......)*** dient nur dazu, das "Standard-Aussehen" dieses Menüs herzustellen, er wird im Programm (vorläufig!) noch keine Wirkung haben.

Die nächste Änderung der Resource-Gabel betrifft das Menü **Bearbeiten**, dessen oberer Teil nun ebenfalls das Standard-Aussehen eines **Bearbeiten**-Menüs erhält. Die alten Befehle rutschen, getrennt durch einen zusätzlichen Strich, in der Liste der Menü-Punkte nach unten.

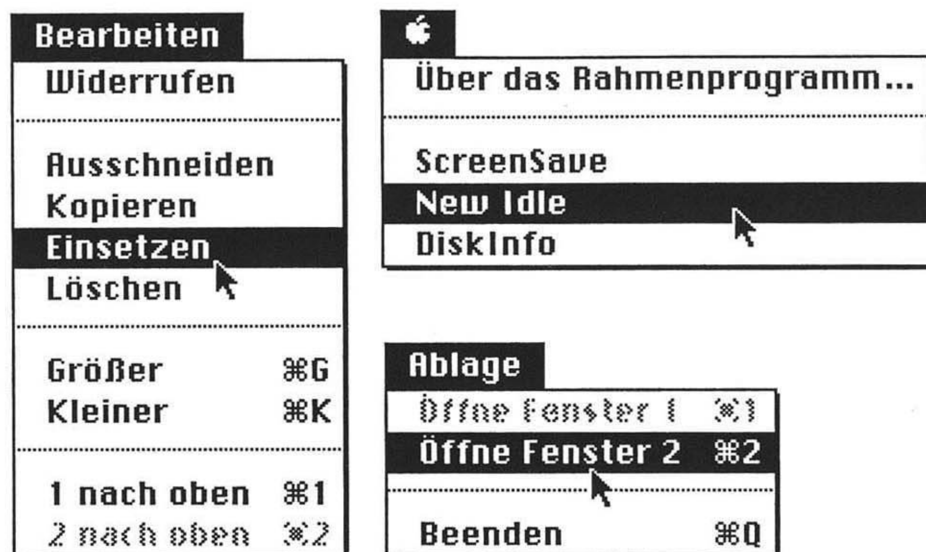


Bild 10 - 4: *Die neuen Menüs des Rahmenprogramms*

Die Änderungen der Menüs haben auch Änderungen in den Konstanten-Definitionen des Rahmenprogramms zur Folge. Insbesondere ändern sich die Nummern der Punkte des Menüs ***Bearbeiten***.

```
CONST
    firstMenu          = 1000;
    lastMenu           = 1002;

    mApfel             = 1000;

    mAblage            = 1001;
        { Punkte bleiben gleich }

    mBearbeiten        = 1002;
        mWider         = 1;
        mAus           = 3;
        mKopie         = 4;
        mEins          = 5;
        mLoesch        = 6;
        { alte Menü-Punkte rutschen nach unten }
        mGroesser      = 8;
        mKleiner       = 9;
        mZeige1        = 11;
        mZeige2        = 12;
```

10.2.3 Änderungen in der Initialisierungsphase

Die einzige Änderung in der Initialisierungsphase des Programms betrifft das Apfel-Menü. Eingelesen und in die Menü-Leiste eingesetzt, wird es ja bereits automatisch durch die Änderungen der entsprechenden Konstanten (**firstMenu**). Zusätzlich müssen aber noch die Namen der zur Verfügung stehenden Schreibtisch-Utensilien an das Menü, so wie es aus der Resource-Datei kommt, angehängt werden.

```
PROCEDURE InitProg;
VAR
    i:              INTEGER;
BEGIN
    MoreMasters;
    MoreMasters;
    MoreMasters;
    MoreMasters;

    FOR i:= firstMenu TO lastmenu DO
        menu[i] := GetMenu(i);

    AddResMenu(menu[mApfel],'DRVR'); { NEU ! }

    FOR i:= firstMenu TO lastmenu DO
        InsertMenu(menu[i],0);
    DrawMenuBar;

    { Der Rest bleibt unverändert }

END;{InitProg}
```

Nachdem alle Menüs aus den Resourcen eingelesen wurden, braucht nur **AddResMenu(...)** aufgerufen werden, und alle Resource-Namen von Resourcen des Typs 'DRVR' werden im Apfel-Menü aufgelistet. Damit kann der Benutzer bereits eine Schreibtisch-Utensilie auswählen. Ohne die weiteren Programmänderungen hat diese Auswahl aber keine Folgen.

10.2.4 Unterstützung periodischer Aktivitäten

Eine der Anforderungen, die Schreibtisch-Utensilien an Programme stellen, ist auch die Unterstützung von periodischen Aktivitäten. So oft wie möglich – mindestens 60mal pro Sekunde — sollte jedes Programm deshalb **SystemTask** aufrufen. Eine Stelle, die ideal für diesen Aufruf geeignet ist, ist die Prozedur **BearbeiteEreig**. Auch sie sollte ja so oft wie möglich durchlaufen werden, um so schnell wie möglich auf jedes Ereignis reagieren zu können.

```
PROCEDURE BearbeiteEreig;
VAR
   ch:      CHAR;
   menuErg: LONGINT;
BEGIN
   SystemTask;      { NEU !}
   IF GetNextEvent(everyEvent,dasEreignis) THEN
   WITH dasEreignis DO
      BEGIN

      { Der Rest bleibt unverändert }

      END;
END;{BearbeiteEreig}
```

10.2.5 Änderungen in der Prozedur MausKlick

In der Prozedur **MausKlick** des Rahmenprogramms wird zunächst über die Funktion **FindWindow** festgestellt, in welche Region auf dem Bildschirm der Maus-Klick getroffen hat. In einer großen CASE-Anweisung wird danach, je nachdem, welches Ergebnis **FindWindow** lieferte, der Maus-Klick weiterverarbeitet.

Ein Fall der CASE-Anweisung war allerdings bis jetzt nicht bearbeitet worden: **inSysWindow**. **InSysWindow** bedeutet aber gerade, daß der Maus-Klick das Fenster einer Schreibtisch-Utensilie getroffen hat. Für die Behandlung solcher Maus-Klicks ist der DeskManager zuständig. **MausKlick** reicht deshalb den entsprechenden Event — der ja in der globalen Variablen **dasEreignis** immer zur Verfügung steht — an die Prozedur **SystemClick** des DeskManagers weiter.

```
PROCEDURE Mausklick(wo: Point);
VAR
    gebiet:       INTEGER;
    fenster:      WindowPtr;
    menuErg:      LONGINT;
BEGIN
    gebiet := FindWindow(wo,fenster);
    CASE gebiet OF

        { Alle anderen Fälle bleiben unverändert }

        inSysWindow:
            BEGIN
            SystemClick(dasEreignis,fenster);
            END; {inSysWindow}

        { Alle anderen Fälle bleiben unverändert }

        END;{CASE}
END;{Mausklick}
```

10.2.6 Änderungen in der Prozedur BeendeProgramm

Durch die Fenster von Schreibtisch-Utensilien kann das Rahmenprogramm jetzt eine ganze Reihe von Fenstern offen haben, wenn der Benutzer den Befehl ***Beenden*** aufruft. Es sieht einfach netter aus, wenn diese Fenster auch nacheinander geschlossen werden, bevor das Programm wirklich verlassen wird. **BeendeProgramm** enthält deshalb eine entsprechende Änderung.

```
PROCEDURE BeendeProgramm;
BEGIN
    SetCursor(GetCursor(4)^^);
    WHILE (FrontWindow <> NIL) DO  { NEU ! }
        SchliesseFenster(FrontWindow);
END; {BeendeProgramm}
```

BeendeProgramm ruft nun einfach immer **SchliesseFenster** für das jeweils vorderste Fenster auf, so lange, bis **FrontWindow** keine offenen

Fenster mehr auf dem Bildschirm findet (NIL ergibt). Diese kleine Änderung macht nicht viel Mühe, sieht aber effektvoll aus.

Die Prozedur **SchliesseFenster** wird im Moment aber nur mit Fenstern fertig, deren **windowKind**-Feld (des entsprechenden WindowRecords) den Wert **grafKind** trägt. Sie muß also abgeändert werden, damit sie auch fähig ist, Fenster von Schreibtisch-Utensilien zu schließen.

10.2.7 Änderungen in der Prozedur SchliesseFenster

```
PROCEDURE SchliesseFenster(fenster: WindowPtr);
BEGIN
    IF (GetWKind(fenster) = grafKind) THEN
       BEGIN

       { Bleibt unverändert }

       END
    ELSE             { wahrscheinlich ein DA-Fenster }
       BEGIN
       IF (GetWKind(fenster) < 0) THEN
          CloseDeskAcc(GetWKind(fenster));
       END;
END; {SchliesseFenster}
```

SchliesseFenster erkennt die Fenster von DAs an einem negativen Ergebnis von **GetWKind**. Dieser Wert ist gleich der Zahl, die **OpenDeskAcc** zurückgibt, nachdem sie eine Schreibtisch-Utensilie geöffnet hat. Es ist die Referenznummer (**refNum**) eines DAs. Mit dieser Zahl kann die Schreibtisch-Utensilie geschlossen werden, was üblicherweise dann auch ein Schließen des entsprechenden Fensters bedeutet.

10.2.8 Änderungen in der Prozedur DoCommand

In **DoCommand** muß jetzt immer eine Schreibtisch-Utensilie geöffnet werden, wenn der Benutzer einen Befehl aus dem Apfel-Menü auswählt, der unter dem Strich liegt, der den ersten Befehl von den Namen der zur Verfügung stehenden Schreibtisch-Utensilien trennt. Der erste Befehl im Apfel-Menü hat noch keine Wirkung.

Eine zweite Änderung in **DoCommand** betrifft das Menü ***Bearbeiten***, in dem ja jetzt auch die Standard-Befehle eines ***Bearbeiten***-Menüs (***Ausschneiden***, ***Kopieren*** etc.) auftauchen. Diese Befehle müssen eventuell an eine Schreibtisch-Utensilie weitergereicht werden.

```
PROCEDURE OeffneDA(daName: Str255);
BEGIN
    { Wird weiter unten erläutert }
END; {OeffneDA}

PROCEDURE DoCommand(menuID,punkt: INTEGER);
VAR
    daName:  Str255;  { NEU ! }
BEGIN
    HiliteMenu(0);
    HiliteMenu(menuID);
    CASE menuID OF
        mApfel:           { NEU ! }
            BEGIN
            IF punkt > 2 THEN
                BEGIN
                GetItem(
                    menu[mApfel],
                    punkt,
                    daName);
                OeffneDA(daName);
                END;
            END;{mApfel}


        mAblage:
            BEGIN
            { bleibt unverändert }
            END;{mAblage}
```

```
        mBearbeiten:
            BEGIN
            CASE punkt OF
               mWider,
               mAus,
               mKopie,
               mEins,
               mLoesch:
                  IF NOT SystemEdit(punkt-1) THEN
                       { passiert nichts };

               { Andere Fälle bleiben unverändert }

               END; { CASE punkt }
            END;{mBearbeiten}
        END; {CASE menuID}
    HiliteMenu(0);
END; {DoCommand}
```

Das Öffnen einer Schreibtisch-Utensilie ist keine ganz triviale Sache. Insbesondere sollte vor dem Öffnen geprüft werden, ob das entsprechende Mini-Programm überhaupt noch in den Speicher paßt. Deshalb wird dieser Vorgang in eine andere Prozedur namens **OeffneDA** ausgelagert, die **DoCommand** aufruft, sobald es den Namen des zu öffnenden DAs festgestellt hat.

Wählt der Benutzer einen der oberen Punkte aus dem Menü ***Bearbeiten***, so reicht **DoCommand** diesen Befehl mittels **SystemEdit** an den DeskManager weiter. Liefert **SystemEdit** als Ergebnis FALSE, sollte der entsprechende Befehl eigentlich vom Programm behandelt werden. Da das Rahmenprogramm die Befehle aus dem ***Bearbeiten***-Menü aber nicht verwendet, werden sie in diesem Fall ignoriert.

```
PROCEDURE OeffneDA(daName: Str255);
VAR
    res,
    dummy:          Handle;
    size:           LONGINT;
BEGIN
    SetResLoad(FALSE);
    res := GetNamedResource('DRVR',daName);
    size := SizeResource(res);
    SetResLoad(TRUE);

    dummy := NewHandle(size+3072);
    IF (MemError = noErr) THEN
        BEGIN
        DisposHandle(dummy);
        size := OpenDeskAcc(daName);
        END;
END; {OeffneDA}
```

OeffneDA überprüft zunächst, ob die Schreibtisch-Utensilie mit dem Namen **daName** überhaupt in den Speicher passen würde. Hierzu wird die Resource angefordert, die den Code dieser Schreibtisch-Utensilie enthält. Zuvor wird aber **SetResLoad(FALSE)** aufgerufen und so verhindert, daß der ResourceManager diese Resource wirklich in den Speicher einliest. Über die Funktion **SizeResource** ermittelt **OeffneDA** dann die Größe des Codes, schlägt noch 3072 Byte für eventuell benötigte Datenbereiche darauf und versucht, einen Handle dieser Größe anzulegen.

Gelingt dies, spricht alles dafür, daß auch das Öffnen der Schreibtisch-Utensilie klappen würde, und der Handle wird freigegeben. Dann wird die Funktion **OpenDeskAcc** aufgerufen, die ja erneut die DRVR-Resource anfordert und damit den freigegebenen Platz wieder einnimmt. Gelingt das Anlegen eines genügend großen Handles nicht, so kehrt **OeffneDA** unverrichteter Dinge zu **DoCommand** zurück, da ein Öffnen des DAs zu riskant wäre.

10.3 Schlußbemerkung

Wie hoffentlich zu sehen war, ist die Behandlung von Schreibtisch-Utensilien mit Hilfe des DeskManagers relativ mühelos möglich. Kein Macintosh-Programm sollte deshalb darauf verzichten. Alle Änderungen, die sich aus

der Unterstützung von Schreibtisch-Utensilien im Rahmenprogramm ergaben, sind zudem weitestgehend übertragbar und bedeuten somit keine zusätzliche Programmierarbeit für neue Programme.

Das Rahmenprogramm in der vorliegenden Form hat allerdings noch einen kleinen Schönheitsfehler. Sobald die Maus-Taste innerhalb eines der beiden Grafik-Fenster gedrückt wird und danach festgehalten wird, kehrt das Programm unter Umständen längere Zeit nicht in die Haupt-Ereignis-Schleife zurück. Schreibtisch-Utensilien, deren Anzeige kontinuierliche Auffrischung verlangt, wie der Wecker oder das Kontrollfeld, erhalten während dieser Zeit keine Gelegenheit zu solchen Auffrischungen – **SystemTask** wird nicht oft genug aufgerufen.

Der Leser möge sich bitte selbst eine Lösung für dieses Problem überlegen. Es gibt mehrere, von denen einige simpel und andere recht ausgefeilt sind und (allgemeingültige) Änderungen an mehreren Stellen des Programms verlangen.

11 Dialoge – die vierte Ausbaustufe

Das letzte Kapitel dieses Buches beschreibt die Möglichkeiten zur Program - mierung von formularähnlichen Eingabemasken, sog. "Dialogen", auf dem Macintosh. Dieses Gebiet ist deshalb in die Auswahl der behandelten Themen mit aufgenommen worden, weil Dialoge zwar nicht zur unbedingt notwendigen Grundausstattung eines Programms gehören, aber doch von fast jedem nichttrivialen Programm genutzt werden.

Der DialogManager, der die Behandlung von Dialogen auf dem Macintosh regelt, ist ein recht komplexes Paket der ToolBox, das eine Reihe anderer Pakete nutzt, die in dem hier zur Verfügung stehenden Raum leider nicht besprochen werden konnten. Dies bringt an einigen Stellen des folgenden Kapitels ein paar Probleme mit sich, die aber hoffentlich nicht so schwerwiegend sind, als daß dadurch das Verständnis dieses Kapitels unmöglich gemacht würde.

11.1 Dialoge auf dem Macintosh

Bei vielen Befehlen, die man einem Programm erteilt, reicht es nicht aus, nur den Befehl selbst aufzurufen, sondern es müssen noch eine Reihe *Parameter* an das Programm übergeben werden — zusätzliche Texte, Zahlen, Ja-Nein-Entscheidungen usw. Erst dadurch erhält das Programm die nötigen Informationen für die Ausführung des Befehls. Zur Ermittlung dieser Zusatzinformationen werden auf dem Macintosh üblicherweise *Dialoge* eingesetzt. Man kann sich das so vorstellen, daß das Programm einen Befehl erhalten hat, aber noch nicht genügend Informationen zu seiner Ausführung besitzt. Diese Informationen werden nun über den Dialog (unter Mithilfe des Benutzers) beschafft.

Dialoge sind eine Art von Formularen (auf anderen Computern oft auch "Masken" genannt), die der Benutzer ausfüllen muß, bevor er mit der Arbeit fortfahren kann. Sie bieten auf dem Macintosh im Vergleich mit ähnlichen Konstrukten auf anderen Rechnern recht vielfältige Möglichkeiten, sind

einfach auszufüllen und können relativ unanfällig gegen Eingabefehler gemacht werden.



Bild 11 - 1: *Ein komplexer Dialog*

Alle grundlegenden Aspekte von Dialogen handhabt dabei, wie inzwischen schon gewohnt, ein Paket der ToolBox: der **DialogManager**. Trotzdem bleibt bei Dialogen – ganz im Gegensatz zu anderen Paketen der Macintosh-Benutzerschnittstelle – noch relativ viel für das Programm selbst zu tun. Dies hat vor allem damit zu tun, daß Dialoge sehr unterschiedlich ausfallen können und zudem noch sehr stark an die Bedürfnisse des Programms angepaßt werden können.

11.1.1 Komponenten eines Dialogs

Standard-Dialoge bestehen üblicherweise aus den folgenden Elementen:

- statischer Text
- Bilder
- Druckknöpfe
- editierbarer Text
- Schalter

• Statische Texte und Bilder dienen dabei nur zur Information des Benutzers. Sie enthalten Hinweise über den Zustand des Programms und Aufforderungen, wie bei der Bearbeitung der restlichen Komponenten des Dialogs vorzugehen ist.

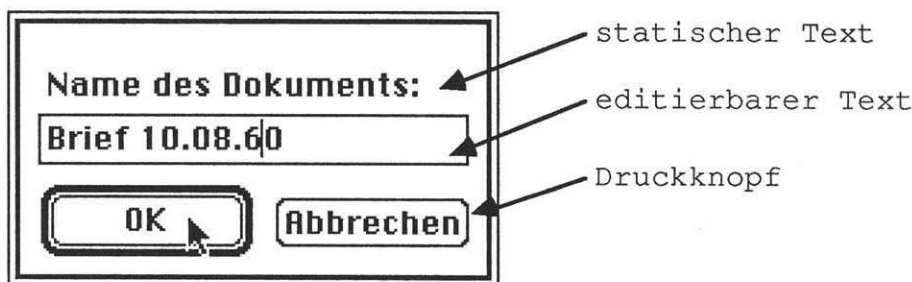


Bild 11 - 2: *Komponenten eines einfachen Dialogs*

Das Drücken eines Druckknopfs schließt normalerweise die Bearbeitung eines Dialogs ab. Ein Druckknopf gilt als gedrückt, wenn die Maus über das Gebiet des Knopfs bewegt wurde, die Maus-Taste gedrückt und losgelassen wurde, während sich der Maus-Cursor nach wie vor über dem Knopf befand. Manche Knöpfe können auch über die Tastatur betätigt werden. Viele Dialoge besitzen einen *Default-Knopf*, der als gedrückt gilt, wenn der Benutzer die Return- oder Enter-Taste drückt. Damit können ohne große Mausbewegungen die vom Programm vorgeschlagenen Voreinstellungen übernommen werden. Der Default-Knopf ist meist durch einen besonderen Rahmen darum hervorgehoben.

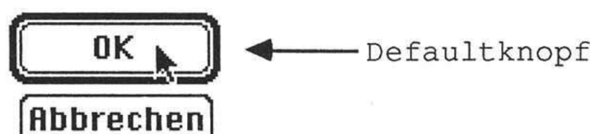


Bild 11 - 3: *Druckknöpfe in einem Dialog*

Die meisten Dialoge haben mindestens zwei Druckknöpfe, die dem Benutzer die Wahl zwischen der Ausführung des mit dem Formular verbundenen Befehls und einem Abbrechen des Befehlsvorgangs lassen. Nachdem der Benutzer also den Befehl aufgerufen hat und den Dialog bearbeitet hat, hat er damit eine letzte Chance, "es sich noch einmal anders zu überlegen".

• Editierbarer Text dient zur Eingabe von Texten und Zahlen in einem Dialog. Editierbare Texte sind im allgemeinen durch einen dünnen Rahmen gekennzeichnet, innerhalb dessen der Text erscheint. In diesem Rahmen kann der Text mit den Hilfsmitteln Maus und Tastatur bearbeitet werden, wie es z.B. aus MacWrite bekannt ist (allerdings nur in einem Zeichensatz und -stil). Die eigentliche Textverarbeitung übernimmt das Paket **TextEdit** in der ToolBox, das im Rahmen dieses Buches nicht näher behandelt wird.

• Schalter dienen dazu, in komfortabler Weise Ja/Nein-Angaben in Formularen zu ermöglichen. Solche Ja/Nein-Angaben (evtl. in mehrfacher Form) dürften jedem, der sich schon einmal mit Formularen herumge - schlagen hat, bekannt sein. Es sind die beliebten Ankreuz-Kästchen. Die Schalter in Dialogen des Macintosh sehen genauso aus. Es handelt sich entweder um kleine Quadrate, die diagonal durchkreuzt werden, oder um kleine Kreise, die mit einem schwarzen Punkt gefüllt werden, wenn dieser Punkt gewählt wird. Sie treten entweder einzeln oder in Gruppen auf.

Treten sie in Gruppen auf, gibt es zwei Varianten davon: eine ausschließliche Auswahl und eine nichtausschließliche (*multiple choice*). Bei einer nichtausschließlichen Auswahl von Schaltern können beliebig viele der Kästchen der Gruppe angekreuzt sein. Die einzelnen Schalter beeinflussen sich gegenseitig nicht. Jeder Medizin-Student dürfte mit dieser Art von Formularen bestens vertraut sein. *Inside Macintosh* nennt sie *ckeckboxes*.



Bild 11 - 4: *Checkboxen*

Die zweite Variante der Dialog-Schalter wird in IM *radio buttons* (Radio-Schalter) genannt. Sie verhalten sich wie manche Schalter-Gruppen auf elektrischen Geräten: drückt man einen der Schalter, springen die anderen der Gruppe heraus. Die Wahl eines der Punkte in der Gruppe ist also ausschließlich ("Kreuzen Sie das Zutreffende an!").



Bild 11 - 5: *Radio-Schalter*

Um dem Benutzer eine zusätzliche Orientierungshilfe zu geben, werden üblicherweise die nichtausschließlichen Schalter-Gruppen immer in Form von (durchkreuzten) Quadraten und die ausschließlichen als (gefüllte) Kreise in Dialogen gezeigt.

Neben Texten, Druckknöpfen und Schaltern können noch eine ganze Reihe anderer Komponenten, z.B. Bilder, Icons, Rollbalken und vom Programm definierte Objekte, in einem Dialog auftauchen. Diese Objekte werden aber

innerhalb dieses Kapitels weiter keine Rolle spielen. Die Behandlung solcher Komponenten-Typen geht nämlich teilweise weit über den Themenkreis hinaus, der in diesem Buch behandelt wird. Näheres dazu ist, wie üblich, in *Inside Macintosh* zu finden.

11.1.2 Grundformen von Dialogen

Dialoge können in mehreren Varianten auftreten. Diese Varianten sind meist durch unterschiedliche Fenster-Typen gekennzeichnet. Prinzipiell kann man unterscheiden zwischen *modalen* und *nichtmodalen* Dialogen.

Modale Dialoge führen auf dem Macintosh etwas ein, was es dort eigentlich überhaupt nicht geben dürfte: einen Modus. Modale Dialoge erscheinen meist in einem Fenster vom Typ **dBoxProc** (vgl. das Kapitel *Fenster-Verwaltung*), einem Fenster mit doppeltem Rahmen. Ein modaler Dialog erscheint immer als oberstes Fenster und muß zuerst durch Druck auf eine der in ihm erscheinenden Tasten beendet werden, bevor die Arbeit weitergehen kann. Nach dem Druck auf den Knopf verschwindet das Fenster üblicherweise. Solange der Dialog aktiv ist (als oberstes Fenster sichtbar ist), befindet sich der Macintosh in einem Modus. Kein Maus-Klick außerhalb des Fensters wird akzeptiert – insbesondere kann keins der anderen Fenster nach vorn geholt werden oder ein Menü-Befehl aufgerufen werden. Jeder Maus-Klick außerhalb des Fensters führt nur zu einem Piepen des Macintosh-Lautsprechers.

Obwohl modale Dialoge einen Modus einführen, was eigentlich ja jedes Programm vermeiden sollte, sind sie oft nötig. Wenn der Benutzer z.B. einen Befehl erteilt hat und vor Ausführung dieses Befehls einfach noch eine Reihe von Daten benötigt werden, kann der Benutzer sinnvollerweise eben nicht an anderer Stelle des Programms weitermachen oder andere Befehle aufrufen. Selbst wenn dies möglich wäre, würde es eventuell den Zustand des bearbeiteten Dokuments verändern und damit den ursprünglich erteilten Befehl, der zum Erscheinen des Dialogs führte, sinnlos machen. Modale Dialoge schützen den Benutzer insofern vor seinen eigenen Fehlern.

Nichtmodale Dialoge werden in anderen Fällen eingesetzt. Vor der Ausführung vieler Befehle kann es sinnvoll sein, die benötigten Daten in Form eines Formulars zu erfragen. Wenn der entsprechende Befehl aber typischerweise mehrfach hintereinander benötigt wird und zudem jederzeit benötigt werden kann, ist es wenig sinnvoll, dieses Formular als modalen Dialog auszuführen.

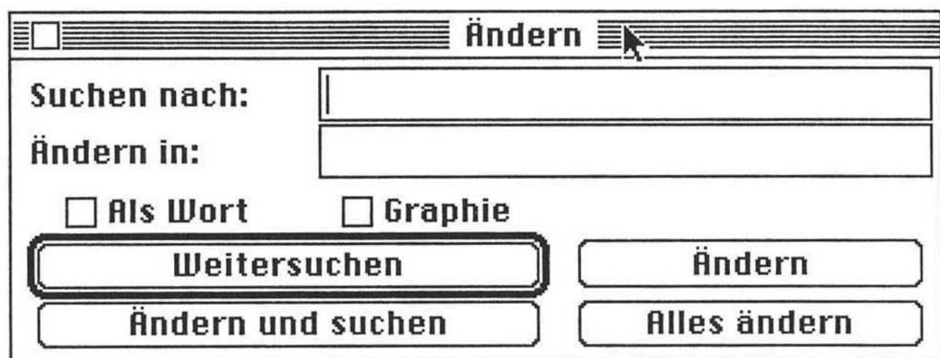


Bild 11 - 7: *Ein nichtmodaler Dialog*

Eine besonders simple Form von (modalen) Dialogen sind Fehlermeldungen und Warnungen (in *Inside Macintosh* unter dem Oberbegriff *Alerts* zusammengefaßt). Auch sie werden auf dem Macintosh üblicherweise in Form von (sehr simplen) Formularen realisiert. Der Benutzer hat bei diesen meist nur die Wahl zwischen verschiedenen Möglichkeiten des Fortfahrens (bei Warnungen), bzw. er muß nur bestätigen, daß er die Meldung gelesen hat (bei Fehlermeldungen).

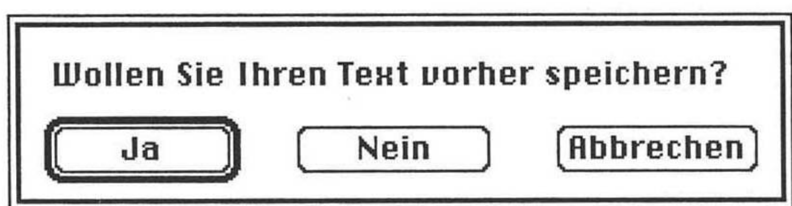


Bild 11 - 7: *Eine System-Meldung*

11.1.3 Operationen und Datentypen des DialogManagers

Der DialogManager stellt nur eine kleine Gruppe von Funktionen zur Verfügung, die für die meisten Programmierer von Interesse sind. Zusätzlich werden allerdings noch einige Operationen erläutert, als wären es Operationen der ToolBox, die aber im Programm selbst realisiert werden müssen. Es handelt sich um Operationen, mit denen Schalter manipuliert werden können. Zum Abfragen und Ändern von Schaltern benötigt man eigentlich den ControlManager, ein Paket der ToolBox, das im Rahmen dieses Buches nicht näher erläutert wird. Er wird deshalb ein wenig umgangen.

Ein Dialog wird im wesentlichen durch den folgenden Datentyp beschrieben:

```
TYPE
    DialogRecord =
    RECORD
        window:       WindowRecord;
        items:        Handle
        textH:        Handle;
        editField:    INTEGER;
        editOpen:     INTEGER;
        aDefItem:     INTEGER;
    END; {DialogRecord};

    DialogPtr   =  WindowPtr;
    DialogPeek  =  ^DialogRecord;
```

Das erste Feld **window** eines DialogRecords ist ein WindowRecord. Es enthält natürlich den WindowRecord desjenigen Fensters, innerhalb dessen der Dialog dargestellt wird.

Das Feld **items** verweist auf die Liste der DialogKomponenten (Texte, Knöpfe, Schalter etc.), die in Form eines verschiebbaren Datenblocks im Heap verwaltet wird. Das genaue Format dieses Datenblocks kann nicht in Form einer Pascal-Typ-Deklaration beschrieben werden und ist – von Ausnahmefällen abgesehen – auch nicht von Interesse.

TextH, **editField** und **editOpen** sind zur internen Verwendung für den DialogManager bestimmt und werden im allgemeinen vom Programmierer nicht verwendet.

Das Feld **aDefItem** enthält die Nummer desjenigen Druck-Knopfes, der als gedrückt gilt, wenn die Return- oder Enter-Taste gedrückt wird, solange dieser Dialog als oberstes Fenster erscheint. Auch dieses Feld wird im allgemeinen nicht vom Programmierer verwendet.

Die Typen DialogPtr und DialogPeek dienen, wie schon von Fenstern bekannt, zum Ansprechen des DialogRecords – die meisten Operationen des DialogManagers erwarten Parameter vom Typ DialogPtr.

Eine Variable vom Typ DialogPtr kann dabei verwendet werden wie eine Variable vom Typ WindowPtr, was oft sehr praktisch für die Manipulation von Dialogen als Ganzes ist. Eine Variable vom Typ DialogPeek hingegen dient der direkten Manipulation der Felder der DialogRecords, was nahezu nie nötig ist.

Dialoge werden üblicherweise komplett als Resourcen in die Resource-Gabel eines Programms gelegt. Resourcen vom Typ 'DITL' (Dialog-Item-List) enthalten Beschreibungen für die Komponenten des Dialogs, also die einzelnen Texte (statisch und editierbar), Druck-Knöpfe, Schalter etc. Resourcen vom Typ 'DLOG' enthalten die Beschreibung für das Aussehen des Dialog-Fensters und enthalten zusätzlich einen Verweis auf die Komponenten, die in diesem Fenster dargestellt werden sollen (eine Resource vom Typ 'DITL').

Aus den DLOG-Resourcen macht der DialogManager dann später DialogRecords, und die DITL-Resourcen werden zu den Datenblöcken im Heap, auf die das Feld **items** des DialogRecords verweist. In diesen Datenblöcken liegen die Beschreibungen für die einzelnen Dialog-Komonenten einfach hintereinander. Eine bestimmte Komponente kann angesprochen werden, indem man ihre Nummer in dieser Liste angibt. Das RMaker-Format für diese beiden Resource-Typen wird weiter unten bei der Prozedur **GetNewDialog** noch genau beschrieben.

Nun aber zu den Operationen des DialogManagers:

```
FUNCTION NewDialog
            (dStorage:   Ptr;
             bounds:     Rect;
             title:      Str255;
             visible:    BOOLEAN;
             procID:     INTEGER;
             behind:     WindowPtr;
             goAwayFlag: BOOLEAN;
             refCon:     LONGINT;
             items:      Handle
            ): DialogPtr;
```

Wer sich die Parameterliste von **NewDialog** genauer anschaut, wird feststellen, daß sie nahezu identisch mit der von **NewWindow** ist (vgl. das Kapitel *Fenster-Verwaltung*). Dies ist auch nicht verwunderlich, da auch Dialoge im wesentlichen Fenster sind, in denen eben nur ein Dialog gezeigt wird. Welcher Dialog dies ist, wird durch den letzten Parameter bestimmt. Die Komponenten des Dialogs müssen in Form der Item-Liste **items** direkt an **NewDialog** übergeben werden. Dies ist aber relativ unüblich und wird nur in den seltensten Fällen wirklich benötigt.

Wie schon an anderen Stellen ist es auch hier "guter Stil", Dialog-Fenster und die Dialog-Komponenten in der Resource-Gabel abzulegen. Deshalb wird die folgende Prozedur auch weit häufiger benötigt.

```
FUNCTION GetNewDialog
            (dialogID:   INTEGER;
             dStorage:   Ptr;
             behind:     WindowPtr
            ): DialogPtr;
```

Ähnlich wie **NewDialog**, erzeugt **GetNewDialog** ein neues Dialog-Fenster, in dem ein bestimmter Dialog gezeigt wird. Nur werden hier die Beschreibung des Fensters und die Liste der Dialog-Komponenten aus zwei Resourcen gelesen. Die Resource vom Typ 'DLOG' mit dem Schlüssel **dialogID** enthält die Beschreibung des Fensters und enthält zusätzlich noch die Nummer einer Resource vom Typ 'DITL', die die Komponenten des

Dialogs enthält. Die anderen Parameter haben dieselben Funktionen wie dieselben Parameter bei **NewDialog** (oder **NewWindow**).

GetNewDialog legt übrigens eine Kopie der entsprechenden DITL-Resource an, bevor ein Handle darauf im Feld **items** des DialogRecords installiert wird! Diese Resource kann also problemlos für mehrere verschiedene Dialoge verwendet werden, die alle dieselben Dialog-Komponenten zeigen. Sie kann deshalb auch gefahrlos purgeable gemacht werden.

Resourcen des Typs 'DITL' und 'DLOG' werden üblicherweise wohl kaum mit dem RMaker erzeugt werden. Dieses Programm ist für solche visuellen Resourcen, wie es Dialoge sind, kaum geeignet. Geigneter sind für dieses Anwendungsgebiet die Resource-Editoren *REdit* und *ResEdit*. Trotzdem wird im folgenden das RMaker-Format für diese beiden Typen beschrieben, da es den Inhalt der entsprechenden Resourcen sehr gut widerspiegelt. Das Format für die Beschreibung der beiden Typen ist dasselbe, wie es im Kapitel *Resourcen* vorgestellt wurde.

```
TYPE DLOG
ResName,ResID(ResAttr)
DialogTitel
oben links unten rechts
{ Visible , InVisible } { GoAway , NoGoAway }
FensterTyp
RefCon
KomponentenNummer
```

Das Format einer Resource vom Typ 'DLOG' ist also nahezu identisch mit dem einer Resource vom Typ 'WIND' (einer Fenster-Beschreibung; vgl. dazu das Kapitel *Resourcen*). Die einzige neue Zeile der Beschreibung ist die letzte: *KomponentenNummer*. An dieser Stelle muß der Schlüssel der Resource vom Typ 'DITL' eingetragen werden, die die Beschreibungen für die Komponenten des Dialogs enthält, der in diesem Fenster dargestellt werden soll.

```
TYPE DITL
ResName,ResID(ResAttr)
AnzahlKomponenten
```

```
{   StaticText,
    EditText,
    Button,
    CheckBox,
    RadioButton  }  {  Enabled, Disabled  }
oben links unten rechts
KomponentenText
```

....

Eine Resource-Beschreibung vom Typ 'DITL' beginnt natürlich mit dem üblichen Kopf jeder Resource-Beschreibung. Danach kommt die Anzahl der folgenden Komponenten und dann die Komponenten-Beschreibungen selbst. Jede davon besteht aus drei Zeilen.

Die erste Zeile einer Komponenten-Beschreibung enthält einen der Bezeichner **StaticText**, **EditText**, **Button**, **CheckBox** und **RadioButton** und optional noch einen der beiden Bezeichner **Enabled** oder **Disabled** dahinter. **StaticText**, **EditText**, **Button**, **CheckBox** oder **RadioButton** bestimmen, ob es sich bei dieser Komponente um einen statischen Text, editierbaren Text, einen Druckknopf oder einen Schalter (in einer der beiden möglichen Varianten) handelt.

Die Angabe von **Enabled** oder **Disabled** dahinter ist nicht unbedingt nötig. Es handelt sich dabei um ein Flag, das für jede einzelne Komponente angegeben werden kann und die spätere Behandlung dieser Komponente durch den DialogManager beeinflußt. Mehr dazu weiter unten, bei der Beschreibung der Prozedur **ModalDialog**. Wird nichts angegeben, setzt der RMaker automatisch **Enabled** ein. Obwohl sich die Angaben zum Komponenten-Typ in der Beschreibung über mehrere Zeilen erstrecken, nehmen sie in einer wirklichen Resource-Beschreibung natürlich nur eine Zeile ein. Die Zeilenbreite dieses Buches reichte einfach nicht für die verschiedenen Möglichkeiten aus.

Die nächste Zeile einer Komponenten-Beschreibung enthält das Rechteck (in lokalen Koordinaten des Dialog-Fensters), in dem die entsprechende Komponente dargestellt werden soll.

Die letzte Zeile enthält einen Text, der in der Komponente erscheinen soll. Bei statischen Texten wird dieser Text innerhalb des oben angegebenen Rechtecks gezeichnet. Ist er zu lang, wird er hinten abgeschnitten. Bei editierbaren Texten erscheint dieser Text als erster Vorschlag, den der Benutzer dann abändern kann. Bei Druckknöpfen wird dieser Text zum Titel des Knopfes, und bei Schaltern erscheint er als Erläuterung rechts daneben.

```
PROCEDURE CloseDialog (whichDialog: DialogPtr);
PROCEDURE DisposDialog(whichDialog: DialogPtr);
```

CloseDialog und **DisposDialog** entsprechen den beiden Prozeduren **CloseWindow** und **DisposWindow** und müssen ähnlich verwendet werden. Sie dienen beide dazu, einen Dialog samt seinem Fenster vom Bildschirm zu entfernen und die damit zusammenhängenden Datenstrukturen freizugeben. **CloseDialog** sollte aufgerufen werden, wenn der Platz für den DialogRecord, auf den **whichDialog** verweist, vom Programm reserviert wurde und an **NewDialog** oder **GetNewDialog** als Parameter **dStorage** übergeben wurde. **DisposDialog** sollte aufgerufen werden, wenn dieser Platz vom WindowManager reserviert wurde und für **dStorage** NIL übergeben wurde.

Beide Prozeduren geben sämtliche anderen dynamisch angelegten Datenstrukturen, die mit dem DialogRecord bzw. dessen Fenster zusammenhängen, frei. Dazu gehören zum Bespiel die verschiedenen Regionen eines Fensters und die Komponenten des Dialogs im Feld **items** des DialogRecords.

Möchte man einen Dialog nicht komplett löschen, sondern ihn nur vom Bildschirm verschwinden lassen, indem man ihn unsichtbar macht, kann man dies mit der Prozedur **HideWindow** des **WindowManagers** tun. Allen Operationen des WindowManagers, die einen WindowPtr erwarten, kann man problemlos einen DialogPtr übergeben (umgekehrt gilt dies natürlich nicht !).

```
PROCEDURE ModalDialog
            (filterProc: Ptr;
             VAR itemHit: INTEGER);
```

ModalDialog ist eine der wichtigsten Prozeduren des DialogManagers überhaupt. Sie übernimmt die komplette Bearbeitung eines Dialogs, der dazu aber im obersten Fenster liegen muß. **ModalDialog** nimmt also an, daß **FrontWindow** einen DialogPtr ergibt. **ModalDialog** ruft selbst

GetNextEvent auf und bearbeitet alle Ereignisse, die von dieser Operation gemeldet werden, selbst, soweit das möglich ist. Hierzu gehört das Tippen von Text und die Bearbeitung dieses Textes mit der Maus, das Anklicken von Knöpfen und Schaltern usw.

Dieser Aufruf von **GetNextEvent** durch **ModalDialog** macht einen modalen Dialog erst zum Modus. Dadurch, daß der DialogManager die Ereignisse bearbeitet, wird im allgemeinen anders auf sie reagiert, als wenn das Programm sie bearbeiten würde. Maus-Klicks außerhalb des Dialogfensters werden z.B. generell ignoriert bzw. haben nur einen Ton aus dem Lautsprecher des Macintosh zur Folge.

Sobald ein Druckknopf oder Schalter betätigt wird, kann der **DialogManager** natürlich nicht wissen, welche Bedeutung dies für das Programm hat. Das Programm muß in diesem Fall eingreifen und kann auch in anderen Fällen eingreifen. Hierfür sind in ModalDialog zwei "Haken" vorgesehen, an denen das Programm ansetzen kann. Sie verbergen hinter den beiden Parametern **filterProc** und **itemHit**.

Hinter dem Parameter **filterProc** steckt ein recht fortgeschrittenes Konzept, das hier nicht weiter behandelt werden soll. Nur soviel: für **filterProc** kann ein Zeiger auf eine Funktion übergeben werden (was in Standard-Pascal eigentlich nicht geht). Dieser Funktion wird jeder Event übergeben, bevor **ModalDialog** ihn selbst bearbeitet. Diese Funktion kann nun selbst auf den Event reagieren oder den Event modifizieren und somit **ModalDialog** trickreich beeinflussen. Näheres aber dazu in *Inside Macintosh*. Für die meisten Anwendungen und das Rahmenprogramm, wie es hier vorgestellt wurde, reicht der zweite "Haken" vollkommen aus.

Der VAR-Parameter **itemHit** wird jedesmal, wenn von **ModalDialog** ein Event empfangen wurde, der eine der Komponenten des Dialogs betrifft, auf die Nummer dieser Komponente gesetzt. Ein Maus-Klick innerhalb eines Druckknopfes führt also z.B. dazu, daß **ModalDialog** beendet wird und **itemHit** die Nummer dieses Knopfes enthält. Das Programm kann nun auf diesen Knopfdruck reagieren und den Dialog z.B. beenden. Oder es kann irgendwelche Änderungen an den Dialog-Komponenten vornehmen und danach **ModalDialog** erneut aufrufen. Da Dialoge normalerweise durch einen Knopf-Druck beendet werden, sieht die Verwendung von **ModalDialog** im Programm deshalb ungefähr so aus wie folgt:

```
REPEAT
    ModalDialog(NIL,komponente);
    CASE komponente OF
        .........
    END; { CASE }
UNTIL komponente gleich einem bestimmten Knopf;
```

Und damit ModalDialog nicht unnötigerweise zurückkehrt, wenn z.B. ein Mausklick eine Komponente getroffen hat, bei der ein Maus-Klick überhaupt keine Rolle spielt, können Komponenten mit dem RMaker oder einem Resource-Editor **disabled** werden. **ModalDialog** kehrt nur dann zurück, wenn ein Ereignis eingetreten ist, das eine Komponente betrifft, die **enabled** ist. (Vergleiche hierzu das Format für die Komponente-Beschreibung, bei der Erläuterung von DITL-Resourcen.) Dies bedeutet allerdings auch, daß in einem Dialog wenigstens eine Komponente **enabled** sein muß, wenn er mit **ModalDialog** bearbeitet werden soll – sonst kehrt **ModalDialog** nämlich nie an das aufrufende Programm zurück.

Statische Texte und editierbare Texte können normalerweise **disabled** werden, während Knöpfe und Schalter **enabled** sein müssen, um auf ihre Betätigung reagieren zu können.

```
FUNCTION  IsDialogEvent
                    (event:  EventRecord): BOOLEAN;
```

Wenn ein Programm nichtmodale Dialoge einsetzt, sollte es, nachdem es mit **GetNextEvent** einen neuen Event empfangen hat, immer mit **IsDialogEvent** prüfen, ob es sich um einen Event handelt, der für einen Dialog bestimmt ist, z.B. ein Maus-Klick in ein Dialog-Fenster oder ein Tastendruck, wenn ein Dialog-Fenster das oberste ist. Ergibt **IsDialogEvent** TRUE, ist es ein Ereignis, das der DialogManager bearbeiten sollte und muß diesem mit **DialogSelect** (s.u.) übergeben werden. Ergibt **IsDialogEvent** FALSE, kann der Event vom Programm bearbeitet werden, wie üblich.

```
FUNCTION   DialogSelect
                (event:           EventRecord;
                 VAR theDialog:   DialogPtr;
                 VAR itemHit:     INTEGER
                ): BOOLEAN;
```

Nachdem mit **IsDialogEvent** festgestellt wurde, daß ein Ereignis für den DialogManager bestimmt ist, sollte **DialogSelect** aufgerufen werden. **DialogSelect** liefert TRUE, wenn dieses Ereignis im Dialog eine Wirkung hat, die für das Programm interessant ist, z.B. ein Maus-Klick in einem Druckknopf, und sonst FALSE.

Ergibt **DialogSelect** TRUE, kann man den beiden VAR-Parametern **theDialog** und **itemHit** entnehmen, welcher Dialog und welche Komponente vom Ereignis **event** betroffen wurden. Das Programm kann dann entsprechend darauf reagieren. Ergibt **DialogSelect** FALSE, können der Event und die beiden VAR-Parameter ignoriert werden.

DialogSelect arbeitet alle üblichen Ereignis-Typen in der korrekten Weise ab, **updateEvts** und **activateEvts** für Dialog-Fenster, **mouseDown**-Events innerhalb von Dialog-Fenstern, **keyDown** und **autoKey**-Events, wenn ein Dialog-Fenster oben liegt, usw. Das Programm braucht sich um solche Ereignisse, sofern sie Dialog-Fenster betreffen, überhaupt nicht mehr zu kümmern.

KeyDown-Events mit gedrückter Befehls-Taste (Tastatur-Äquivalente für Menü-Befehle) werden allerdings nicht erkannt. Wenn ein Programm auch dann Tastatur-Äquivalente für Menü-Befehle zulassen will, wenn ein Dialog-Fenster an oberster Stelle liegt, sollte es zunächst diesen Fall überprüfen, bevor es einen Event an **DialogSelect** weitergibt und gegebenenfalls das Ereignis selbst bearbeiten.

```
FUNCTION   Alert
                (alertID:     INTEGER;
                 filterProc:  Ptr
                ): INTEGER;
```

Während **ModalDialog** dem Programm schon ziemlich viel Arbeit mit Dialogen abnimmt, ist **Alert** noch etwas einfacher zu "bedienen". Alerts sind besonders simple Spezialfälle von modalen Dialogen. Sie dienen dazu,

allgemeine Meldungen oder Fehlermeldungen an den Benutzer zu übermitteln oder sehr einfache Entscheidungen zu erfragen.

Sie bestehen meist nur aus einem oder einigen wenigen statischen Texten und mindestens einem Druckknopf. Beendet wird ein Alert meist durch den Druck auf einen der Knöpfe. (Bild 11-7 zeigt einen typischen Alert.) **Alert** kehrt deshalb im allgemeinen auch erst dann zurück, wenn der Benutzer einen Knopf gedrückt hat, und liefert als Funktionswert die Nummer des gedrück - ten Knopfes zurück. **FilterProc** hat dieselbe Bedeutung wie bei **ModalDialog** und soll hier nicht weiter erläutert werden.

Bei Alerts hat der Programmierer keinen Einfluß auf das Aussehen des Fensters, in dem dieser gezeigt wird. Es sind immer Fenster mit doppeltem Rahmen (Typ **dBoxProc**), in denen Alerts erscheinen. Die Resource-Beschreibung für Alerts hat das folgende Format.

```
TYPE ALRT
ResName,ResID(ResAttr)
oben links unten rechts
KomponentenNummer
AlertStufen
```

Genau wie bei Resourcen vom Typ 'DLOG', müssen auch hier Größe und Position des Fensters angegeben werden; über den Fenster-Typ, die Sichtbarkeit des Fensters etc. entscheidet aber der DialogManager bzw. die Prozedur **Alert** selbst. Genauso muß an der Stelle *KomponentenNummer* auch der Schlüssel für die Resource vom Typ 'DITL' angegeben werden, die die Liste der zu zeigenden Komponenten enthält. Die letzte Zeile der Resource-Beschreibung enthält die Spezifikation der vier Alert-Stufen. Diese Angabe muß in Form einer 16-Bit-Zahl gemacht werden, die in sehr gepackter Form die folgenden Angaben enthält:

- wie oft der Lautsprecher bei einem Alert piepen soll (kein- bis dreimal)
- ob der Alert überhaupt gezeigt werden soll
- welcher Knopf des Alerts der Default-Knopf ist

Diese drei Angaben können für jede der vier Stufen eines Alerts verschieden sein. Bei Fehlermeldungen ist es vielleicht sinnvoll, beim ersten Auftreten des Fehlers zunächst nur den Lautsprecher piepen zu lassen. Beim zweiten Mal piept der Lausprecher zweimal, beim dritten Mal dreimal und beim vierten Mal auch dreimal, und zusätzlich wird der Alert das erste Mal gezeigt. Alle folgenden Fehler werden genauso behandelt, wie der vierte. Für den letzten

Alert (bzw. die entsprechende Resource) merkt sich der DialogManager, wie oft er schon aufgerufen wurde und handelt entsprechend der Angabe im Feld *AlertStufen* der Resource-Beschreibung.

Für jede Stufe werden diese Angaben in vier Bits gepackt. Die folgende Beschreibung geht von links nach rechts (von den höherwertigen zu den niederwertigen Bits). Das höherwertigste (linke) Bit bestimmt den Default-Knopf. Ist es gleich 0, ist der Knopf mit der Nummer 1 der Default-Knopf. Ist es gleich 1, ist es der Knopf mit der Nummer 2. Das folgende Bit bestimmt, ob der Alert gezeichnet wird oder nicht (wenn es gleich 0 ist). Die beiden niederwertigsten Bits bestimmen die Anzahl der Töne, die erzeugt werden, wenn der entsprechende Alert aufgerufen wird. Sie können als ganze Zahl mit 2 Bits interpretiert werden (von Null bis drei).

Die Angaben für die vier Stufen werden dann zu einer 16-Bit-Zahl zusammengefaßt, die der RMaker in hexadezimaler Form (also in Form von vier Hexadezimal-Ziffern von '0' bis 'F') erwartet. Die erste Ziffer dieser Zahl entspricht den Angaben für die vierte und alle folgenden Stufen, die zweite denen für die dritte, die dritte denen für die zweite und die vierte Ziffer enthält die Angaben für die erste Stufe des Alerts. Grundkenntnisse in binärer und hexadezimaler Arithmetik sind zum Berechnen dieser vier Ziffern unerläßlich. Wer diese nicht besitzt, kann die entsprechenden Angaben auch mit einem Resource-Editor in komfortabler Weise machen und die Arithmetik diesem Programm überlassen.

```
FUNCTION   StopAlert
                (alertID:    INTEGER;
                 filterProc: Ptr
                ): INTEGER;

FUNCTION   NoteAlert
                (alertID:    INTEGER;
                 filterProc: Ptr
                ): INTEGER;

FUNCTION   CautionAlert
                (alertID:    INTEGER;
                 filterProc: Ptr
                ): INTEGER;
```

Neben dem "normalen" Alert gibt es noch eine Reihe von besonderen Alerts, die durch diese drei Funktionen gezeigt werden können. Sie zeigen zusätzlich zu den Komponenten, die in der Komponentenliste angegeben wurden, noch ein Icon in der linken oberen Ecke (im Rechteck ((10,20),(42,52))) des Fensters, das dem Benutzer helfen soll, die Bedeutung dieses Alerts schneller einzuordnen. Ansonsten werden sie vom Programm aus genauso wie die "normalen" Alerts behandelt, und ihr Resource-Format ist auch dasselbe.



Bild 11 - 8: *Das Stop-Icon*

StopAlert wird normalerweise aufgerufen, um dem Benutzer zu melden, daß eine von ihm veranlaßte Operation gestoppt wurde, da sie zu einem Fehler führte oder führen würde. Es ist eine typische Macintosh-Fehlermeldung. Der entsprechende Dialog hat üblicherweise nur einen Knopf, mit dem der Benutzer die Meldung zur Kenntnis nehmen muß.



Bild 11 - 9: *Das Note-Icon*

Der Aufruf von **NoteAlert** bedeutet normalerweise eine Meldung an den Benutzer, daß eine besonders wichtige oder schwerwiegende Aktion durchgeführt wurde oder wird. Er bedeutet üblicherweise keinen Fehler, sondern nur eine "Bemerkung des Programms". Der entsprechende Dialog hat üblicherweise nur einen Knopf, mit dem der Benutzer die Meldung zur Kenntnis nehmen muß.



Bild 11 - 10: *Das Caution-Icon*

Der Aufruf von **CautionAlert** dient zur Warnung des Benutzers, die üblicherweise aus einer Frage der folgenden Form besteht: "Sie haben den Befehl XXX gewählt, der YYY zur Folge hat. Wollen wirklich ZZZ". Der Benutzer hat nun die Gelegenheit, durch Drücken des entsprechenden Knopfes entweder den Befehl zu bestätigen oder es sich noch einmal anders zu überlegen. Der entsprechende Dialog hat üblicherweise zwei oder drei Knöpfe, mit denen der Benutzer unter den möglichen Alternativen wählen kann. Das Programm kann am Ergebnis der Funktion **CautionAlert** ablesen, welche Entscheidung der Benutzer getroffen hat. Bild 11-7 zeigt einen typischen **CautionAlert**.

Die Icons für alle drei besonderen Arten von Alerts befinden sich in der System-Resource-Gabel. Möchte man in seinem Programm andere Icons an deren Stelle zeigen, so braucht man nur in der Resource-Gabel seines Programms Icons mit denselben Resource-Schlüsseln definieren. Diese Schlüssel sind: 0 für das Stop-Icon, 1 für das Note-Icon und 2 für das Caution-Icon.

Die folgenden Prozeduren dienen zum Abfragen und zur Manipulation von Komponenten von Dialogen (und Alerts). Sie werden üblicherweise dann aufgerufen, wenn der Benutzer auf einen Knopf oder Schalter gedrückt hat (und damit eventuell den Dialog beendet hat).

```
PROCEDURE ParamText(t0,t1,t2,t3: Str255);
```

Mit **ParamText** können auf einfache Weise statische Texte in Dialogen geändert werden. Der eigentliche Text muß dazu einen der vier "Platzhalter" '^0', '^1', '^2' oder '^3' enthalten. **ParamText** ersetzt diese vier Kontroll-Kombinationen von 2 Buchstaben durch die vier Strings, die als Parameter übergeben wurden. Auf diese Weise kann man z.B. nur eine Resource vom Typ 'DITL' für Fehlermeldungen haben, und trotzdem immer verschiedene Meldungen am Bildschirm zeigen. **ParamText** wird sinnvollerweise meist aufgerufen, bevor der entsprechende Dialog oder Alert gezeigt wird, damit sich der Text nicht vor den Augen des Benutzers ändert.

```
PROCEDURE GetDItem
                  (dialog:    DialogPtr;
                   itemNo:    INTEGER;
                   VAR type:  INTEGER;
                   VAR item:  Handle;
                   VAR box:   Rect);
```

GetDItem liefert in **item** einen Handle auf eine Komponente des Dialogs **dialog** zurück. Welche Komponente dies ist, bestimmt die Nummer **itemNo**. Weitere Informationen über diese Komponente können mit einer der noch folgenden Operationen gewonnen werden. Einige Informationen werden aber auch schon in den VAR-Parametern **type** und **box** zurückgeliefert. **Type** informiert über die Art der Komponente und **box** über die Größe und Position der Komponente im Dialog-Fenster. Für **type** sind die folgenden Konstanten definiert:

```
CONST
    ctrlItem  = 4;  {zum Addieren auf andere Konst.}
    btnCtrl   = 0;  {Druckknopf}
    chkCtrl   = 1;  {Check-Box-Schalter}
    radCtrl   = 2;  {Radio-Button-Schalter}
    resCtrl   = 3;
    statText  = 8;  {statischer Text}
    editText  = 16;{editierbarer Text}
```

Merkwürdigerweise sind für die Typen von Kontrollen (Druckknöpfen und Schaltern) keine Konstanten direkt definiert, sondern eine Kombination zweier Konstanten, die addiert werden müssen. Auf die Konstante für den entsprechenden Typ muß stets **ctrlItem** aufaddiert werden, um den wirklichen Wert von **type** zu erhalten, den dieser Parameter besitzt, wenn eine solche Komponente mit **GetDItem** abgefragt wird. Druckknöpfe haben also in Wirklichkeit die Typ-Nummer 4, CheckBox-Schalter die Nummer 5 und RadioButton-Schalter die Nummer 6.

Genauso haben Komponenten, die **disabled** sind, dieselben Typ-Nummern wie ihre **enabled**-Formen, auf die noch 128 addiert wurde. Die oben aufgelisteten Konstanten gelten für Komponenten, die **enabled** sind. Ein statischer Text, der **disabled** ist, hätte demnach die Typ-Nummer 136.

```
PROCEDURE GetIText(item: Handle;VAR text: Str255);
PROCEDURE SetIText(item: Handle;    text: Str255);
```

Mit **GetIText** und **SetIText** kann der, bei einer Dialog-Komponenten gezeigte, Text abgefragt und geändert werden. Dies ist derselbe Text, der in der letzten Zeile der Resource-Beschreibung einer Dialog-Komponenten angegeben wird. Bei statischen und editierbaren Texten sind es die Texte selbst, bei Druckknöpfen sind es deren Namen, bei Schaltern die Texte, die neben ihnen erscheinen.

Eine Hauptanwendung für **GetIText** ist z.B. das Lesen derjenigen Texte, die ein Benutzer in den Rahmen eines editierbaren Textes eingegeben hat. Enthält ein Dialog mehr als vier veränderliche statische Texte, reicht **ParamText** nicht mehr aus, die nötigen Veränderungen auszuführen. Hierfür kann dann (etwas umständlicher) **SetItext** verwendet werden.

Die folgenden Prozeduren werden hier beschrieben, als wären sie in der ToolBox implementiert. Sie müssen aber im Rahmenprogramm geschrieben werden, da sie aus unerfindlichen Gründen nicht in der ToolBox sind. Ihre Aufgabe ist die Feststellung des Zustands eines Schalters und die Änderung dieses Zustand (Ein oder Aus bzw. Wahr oder Falsch).

```
FUNCTION   SchalterAn
             (dialog:   DialogPtr;
              schalter: INTEGER
             ): BOOLEAN;
```

SchalterAn liefert genau dann TRUE, wenn die Komponente mit der Nummer **schalter** des Dialogs **dialog** ein Schalter ist, der "An" ist.

```
PROCEDURE  EinSchalter
             (dialog:   DialogPtr;
              schalter: INTEGER
             );
```

EinSchalter setzt die Komponente **schalter** des Dialogs **dialog** auf den Zustand "An" (bzw. "Ein"), falls es sich bei dieser um einen Schalter handelt.

```
PROCEDURE  AusSchalter
             (dialog:   DialogPtr;
              schalter: INTEGER
             );
```

AusSchalter setzt die Komponente **schalter** des Dialogs **dialog** auf den Zustand "Aus", falls es sich bei dieser um einen Schalter handelt.

```
PROCEDURE  KippSchalter
             (dialog:   DialogPtr;
              schalter: INTEGER
             );
```

KippSchalter setzt die Komponente **schalter** des Dialogs **dialog** auf den Zustand "Aus", falls es sich bei dieser um einen Schalter handelt, der vorher "An" war, und auf "An", falls er vorher "Aus" war.

```
PROCEDURE RadioEin
            (dialog:   DialogPtr;
             schalter: INTEGER;
             gruppe:   SetOfSchalter
            );
```

RadioEin setzt die Komponente **schalter** des Dialogs **dialog** auf den Zustand "An" und alle anderen Schalter aus der Menge **gruppe** auf "Aus". Der Typ **SetOfSchalter** ist dabei definiert wie folgt:

```
TYPE
    SetOfSchalter= SET OF 1..256;
```

256 schien eine sinnvolle obere Schranke für die Anzahl von Komponenten in einem Dialog zu sein. Viel mehr Komponenten dürften zu sehr unübersichtlichen Dialogen führen. Eine Variable vom Typ **SetOfSchalter** benötigt auch mit dieser Einschränkung immerhin schon 32 Byte (256 Bit). **RadioEin** wird natürlich im Fall eines Klicks des Benutzers auf einen RadioButton-Schalter benötigt, um in diesem Fall den angeklickten Schalter auf "Ein" zu setzen und alle anderen Schalter der Gruppe auf "Aus".

Da die Prozedur **RadioEin** den Pascal-Set-Konstruktor verwendet, dürfte sie recht schwer auf andere Sprachen übertragbar sein!

11.2 Nutzung des DialogManagers im Rahmenprogramm

Das Rahmenprogramm kennt eigentlich keine sinnvollen Anwendungen für Dialoge und Alerts. Deshalb ist die Verwendung von Dialogen mehr eine Spielerei. Alle drei Arten von Dialogen werden dabei vorgeführt (modale und nichtmodale Dialoge und Alerts) und alle in den vorangegangenen Abschnitten erläuterten Komponenten-Typen.

Der nichtmodale Dialog wird gleich zu Anfang des Programms erzeugt und erscheint als hinterstes Fenster auf dem Bildschirm. Er wird eine Reihe von Schaltern und zwei Knöpfe besitzen. Wenn der Benutzer einen der beiden Knöpfe drückt, wird eine Meldung erscheinen (ein Alert), deren Text abhängig von den Schalterstellungen des nichtmodalen Dialogs ist. Die folgende Abbildung zeigt einen Bildschirm-Ausschnitt des laufenden Programms, nachdem gerade ein Knopf des Dialogs gedrückt wurde:

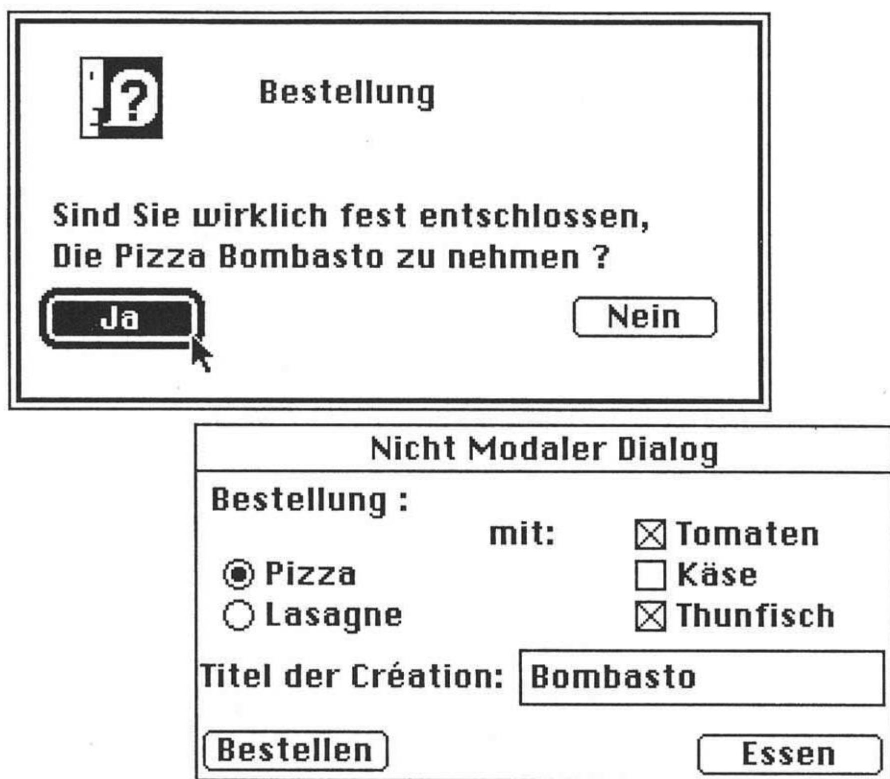


Bild 11 - 11: *Dialoge und Alerts in Aktion*

Der modale Dialog wird ein sehr simpler Dialog sein, der auf den Menü-Befehl **Über das Rahmenprogramm...** hin erscheinen soll, wie es auch bei anderen Programmen üblich ist.

11.2.1 Überblick über die Änderungen

Das Programm wird insgesamt vier verschiedene Dialoge zeigen: einen nichtmodalen Dialog, einen modalen Dialog und einen Alert. Die für den modalen Dialog und die beiden Alerts nötigen Änderungen sind sehr

programmspezifisch und können nur als Beispiel für die Verwendung solcher Dialoge in anderen Programmen dienen. Die für den nichtmodalen Dialog notwendigen Änderungen setzen teilweise "tief unten" im Programm bei der Verarbeitung von Ereignissen an. Sie sind in der hier vorgestellten Form in jedes Programm übertragbar, das nichtmodale Dialoge einsetzt.

11.2.2 Änderungen an der Resource-Datei

In der Resource-Gabel des Programms müssen vor dem Start des Programms die Resourcen vom Typ 'DITL' untergebracht werden, die die Komponenten der verschiedenen Dialoge und Alerts beschreiben und natürlich auch die DLOG- und ALRT-Resourcen, die das Aussehen der Dialoge und Alerts selbst bestimmen. Alle Resourcen, die im folgenden aufgelistet werden, kommen zu den schon in den vorangehenden Kapiteln beschriebenen dazu.

Um die Übersichtlichkeit der folgenden Beschreibung zu erhöhen, wurden jeweils die DITL-Resourcen zusammen mit ihren zugehörigen DLOG- bzw. ALRT-Resourcen aufgelistet. Diese Form der Resource-Beschreibung kann auch in die wirkliche RMaker-Eingabedatei übernommen werden. Wem es übersichtlicher erscheint, der kann aber auch alle DLOG-, DITL- und ALRT-Resourcen jeweils zusammen hinter einem TYPE-Statement auflisten. Dem RMAker ist es egal, ob Resourcen eines Typs alle hintereinander aufgelistet werden oder wild verstreut sind.

Damit der Leser sich auch eine Vorstellung vom Aussehen des "fertigen Produkts" machen kann, werden hinter den Resource-Beschreibungen jeweils auch noch die Dialoge bzw. Alerts abgebildet, wie sie auf dem Bildschirm aussehen sollten.

```
TYPE DLOG
    ,1001
Nicht Modaler Dialog
40 200 160 470
Visible NoGoAway
0
0
1001
```

```
TYPE DITL
    ,1001
11
BtnItem Enabled
100 2 117 74
Bestellen

BtnItem Enabled
100 194 118 266
Essen

StatText Disabled
3 4 19 84
Bestellung :

RadioItem Enabled
32 8 48 72
Pizza

RadioItem Enabled
48 8 64 88
Lasagne

CheckItem Enabled
16 168 32 248
Tomaten

CheckItem Enabled
32 168 48 224
Käse

CheckItem Enabled
48 168 64 256
Thunfisch

EditText Disabled
72 128 88 264


StatText Disabled
72 0 88 120
Titel der Création:
```

```
StatText Disabled
16 112 32 144
mit:
```



Bild 11 - 12: *Ein nichtmodaler Dialog*

```
TYPE ALRT
    ,1003
50 100 190 380
1003
7654

TYPE DITL
    ,1003
5
BtnItem Enabled
104 8 120 64
Ja

BtnItem Enabled
103 211 120 267
Nein

StatText Disabled
16 88 32 160
Bestellung
```

```
StatText Disabled
64 8 80 248
Sind Sie wirklich fest entschlossen,

StatText Disabled
80 8 95 268
Die ^1 ^0 zu nehmen ?
```

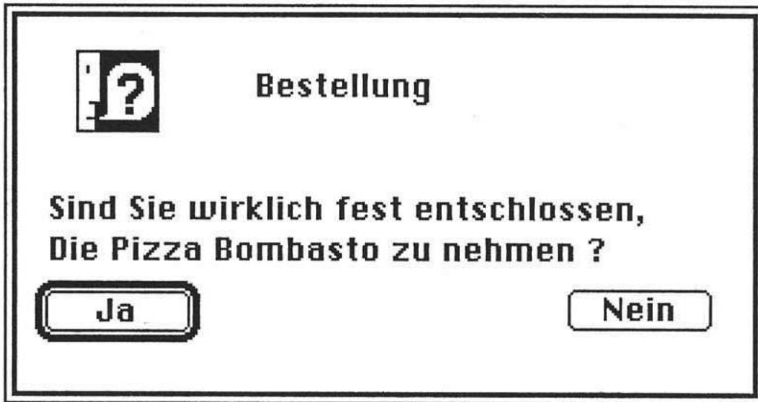


Bild 11 - 13: *Der "Bestellung"-Alert*

```
TYPE ALRT
    ,1004
40 120 190 320
1004
7654

TYPE DITL
    ,1004
4
BtnItem Enabled
111 134 127 190
Danke !

StatText Disabled
64 0 80 192
Wir wünschen guten Appetit
```

```
StatText Disabled
8 128 24 192
Zur ^1 ^0 !

StatText Disabled
8 128 24 192
Mahlzeit
```



Bild 11 - 14: *Der "Mahlzeit"-Alert*

```
TYPE DLOG
    ,1002
Über… -Dialog
40 100 150 340
Visible NoGoAway
3
0
1002

TYPE DITL
    ,1002
3
BtnItem Enabled
80 96 96 152
OK
```

```
StatText Disabled
0 88 16 152
Rahmen4

StatText Disabled
16 56 64 184
Ein Programm, das Dialoge und Alerts demonstriert !
```



Bild 11 - 15: *Über das Rahmenprogramm...*

11.2.3 Änderungen in der Initialisierungsphase

In der Initialisierungsphase des Programms braucht nur eine einzige Änderung durchgeführt werden. Diese ist auch nicht allgemeingültig, sondern sehr programmspezifisch. Der nichtmodale Dialog, der ja schon zu Beginn des Programms erscheinen soll, muß (aus der Resource-Gabel) eingelesen und geöffnet werden.

```
CONST
    idNichtMod   = 1001;
VAR
    nichtModal   : DialogPtr;
```

```
PROCEDURE InitProg;
VAR
    i:              INTEGER;
BEGIN

    { Der Beginn bleibt völlig unverändert }

    nichtModal := GetNewDialog(
                        idNichtModal,
                        NIL,
                        Pointer(-1));

    fenster1   := GetNewWindow(
                        idWin1,
                        NIL,
                        Pointer(-1));
    ......
    { Der Rest bleibt völlig unverändert }
    ......

END;{InitProg}
```

Der nichtmodale Dialog wird als erstes Fenster erzeugt und liegt somit beim Start des Programms hinter allen anderen. Erst danach werden die beiden anderen Fenster aus der Resource-Gabel eingelesen und das erste davon auch auf dem Bildschirm gezeigt.

Sicherheitshalber wird ein Zeiger auf den erzeugten Dialog in einer globalen Variablen gespeichert. Dies ist im Moment zwar nicht unbedingt nötig, da das Programm sowieso nur einen solchen Dialog behandelt, ist für einen eventuellen Ausbau aber ganz sinnvoll.

11.2.4 Änderungen an BearbeiteEreig

Besonders schwerwiegende Änderungen ergeben sich an **BearbeiteEreig**. Die beiden Operationen **IsDialogEvent** und **DialogSelect**, die zur Unterstützung nichtmodaler Dialoge benötigt werden, müssen hier – sofort nachdem ein neues Ereignis empfangen wurde – aufgerufen werden.

```
PROCEDURE DialogEvent
                (dialog:      DialogPtr;
                 komponente: INTEGER);
BEGIN
    { Wird weiter unten näher beschrieben }
END;{DialogEvent}

PROCEDURE BearbeiteEreig;
VAR
    ch:        CHAR;
    menuErg: LONGINT;
    dialog:  DialogPtr;   { NEU !}
    itemHit: INTEGER;     { NEU !}


BEGIN
    IF GetNextEvent(everyEvent,dasEreignis) THEN
    WITH dasEreignis DO
        BEGIN
        fMouse    := (BitAnd(modifiers,128)  <> 0);
        fShift    := (BitAnd(modifiers,512)  <> 0);
        fShLock   := (BitAnd(modifiers,1024)<> 0);
        fOption   := (BitAnd(modifiers,2048)<> 0);
        fCommand := (BitAnd(modifiers,256)  <> 0);
        fActive   := (BitAnd(modifiers,001)  <> 0);

        IF IsDialogEvent(dasEreignis)
        AND NOT ((what=keyDown) AND fCommand) THEN
            BEGIN
            IF DialogSelect(dasEreignis,
                                dialog,
                                itemHit)
            THEN
                DialogEvent(dialog,itemHit);
            END
```

```
        ELSE
            CASE what OF
                ......

                { Der Rest bleibt gleich }

                ......
                END;{CASE};
        END;{WITH}
END;{BearbeiteEreig}
```

Der Anfang von **BearbeiteEreig** bleibt gleich. Nach dem Empfang eines Ereignisses werden zunächst die **modifier**-Bits analysiert. Danach wird aber mit **IsDialogEvent** geprüft, ob es sich bei dem empfangenen Ereignis um eines handelt, für das der DialogManager zuständig ist. Ist das der Fall, so wird es an **DialogSelect** weitergereicht. **DialogSelect** meldet in seinem Funktionsergebniss, ob das Ereignis Änderungen zur Folge hatte, die für das Programm interessant sein könnten. Liefert **DialogSelect** TRUE, so reichen wir einen Zeiger auf den betroffenen Dialog und die Nummer der betroffenen Komponente an die Prozedur **DialogEvent** weiter, die im nächsten Abschnitt näher beschrieben wird.

Liefert **IsDialogEvent** FALSE oder handelt es sich um das Tastatur-Äquivalent eines Menü-Befehls, so wird das Ereignis behandelt, wie schon in den letzten Kapiteln beschrieben. Die zusätzliche Abfrage auf Tastatur-Äquivalente wird deshalb nötig, da diese von **DialogSelect** nicht bearbeitet werden.

Der Leser möge bitte auch bei diesen Änderungen wieder die saubere Trennung in allgemeingültige und programmspezifische Programmteile beachten. **BearbeiteEreig** ist auch in seiner neuesten (und letzten) Form allgemeingültig. Die programmspezifische Bearbeitung von nichtmodalen Dialogen wird an die Prozedur **DialogEvent** ausgelagert, die im folgenden beschrieben wird.

11.2.5 Die Prozedur DialogEvent

Um die Behandlung von Ereignissen, die Dialoge betreffen, soweit wie möglich unabhängig vom Programm realisieren zu können, wird der Teil der Dialog-Verarbeitung, der programmspezifisch ist, in der separaten Prozedur **DialogEvent** realisiert.

```
PROCEDURE Bestellen
                  (name:  Str255;
                   pizza: BOOLEAN;
                   essen: BOOLEAN);
BEGIN
    { Wird weiter unten näher beschrieben }
END;{Bestellen}

PROCEDURE DialogEvent
                       (dialog:      DialogPtr;
                        komponente: INTEGER);
CONST
     butBestell       = 1;
     butEssen         = 2;

     itPizza          = 4;
     itLasag          = 5;
     itTomat          = 6;
     itKase           = 7;
     itThunf          = 8;
     itName           = 9;
VAR
     itTyp:     INTEGER;
     item:      Handle;
     box:       Rect;
     name:      Str255;

BEGIN
    CASE itemHit OF
        butBestell,butEssen:
            BEGIN
            GetDItem(dialog,itName,
                     itTyp,item,box);
            GetIText(item,name);
            Bestellen(name,
                      SchalterAn(dialog,itPizza),
                      (itemHit=butEssen));
            END;
```

```
        itPizza,itLasagne:
            RadioEin(dialog,
                    itemHit,[itPizza,itLasagne]);

        itTomat, itKase, itThunf:
            EinSchalter(dialog,itemHit);
        END; {CASE}
END;{DialogEvent}
```

DialogEvent behandelt in der hier vorgestellten einfachen Form nur einen Dialog. Bei einem Programm, das mehrere nichtmodale Dialoge verwaltet, müßte auch abgeprüft werden, um welchen Dialog es sich handelt, und abhängig davon unterschiedlich verzweigt werden.

Hier besteht **DialogEvent** nur aus einer großen CASE-Anweisung, die nach der "getroffenen" Dialog-Komponente **itemHit** unterscheidet. Wird einer der beiden Knöpfe gedrückt, so wird die Prozedur **Bestellen** aufgerufen, die weiter unten noch detailliert beschrieben wird. **Bestellen** zeigt dann einen der beiden Alerts, die bereits in einem der vorangegangenen Abschnitte abgebildet wurden.

Zur besseren Lesbarkeit des Programmtextes werden für alle wichtigen Komponenten des Dialogs Konstanten definiert, die die Komponenten-Nummer der entsprechenden Komponente enthalten.

Wird einer der Radio-Schalter gedrückt, so wird mit **RadioEin** dieser eingeschaltet und der andere ausgeschaltet. Beide Radio-Schalter können problemlos gemeinsam behandelt werden, da die Variable **itemHit** mit der genauen Komponenten-Nummer des Schalters ja immer zu Verfügung steht. Die Schreibweise **[itPizza,itLasagne]** sollte jedem bekannt sein, der sich schon einmal mit Mengen in Pascal beschäftigt hat. Hierdurch wird die Menge eindeutig beschrieben, die aus den beiden Zahlen **itPizza** und **itLasagne** besteht.

Wird einer der CheckBox-Schalter gedrückt, so wird sein Zustand mit **KippSchalter** umgekehrt. Dies hat keinen Einfluß auf andere Dialog-Komponenten.

DialogEvent demonstriert die Behandlung aller wichtigen Typen von Dialog-Komponenten und ist ansonsten natürlich sehr programmspezifisch. Besonders interessant ist die Feststellung des vom Benutzer eingegebenen Textes für den Namen der Pizza bzw. Lasagne mit **GetIText**.

Es fehlt jetzt nur noch die Beschreibung der Prozedur **Bestellen**, die die Bestellung des Benutzers annimmt und bearbeitet.

```
CONST
        idBestellen = 1003;
        idEssen     = 1004;

PROCEDURE Bestellen
                (name:  Str255;
                 pizza: BOOLEAN;
                 essen: BOOLEAN);
VAR
    itemHit:  INTEGER;
BEGIN
    IF pizza THEN
        ParamText(name,'Pizza','','')
    ELSE
        ParamText(name,'Lasagne','','');

    IF essen THEN
        itemHit := StopAlert(1004,NIL)
    ELSE
        itemHit := CautionAlert(1003,NIL);

END;{Bestellen}
```

Die genaue Aktion von **Bestellen** ist nur abhängig von seinen drei Parametern. Ist **pizza** TRUE, lautet die Bestellung auf eine Pizza, der Titel der Pizza (bzw. Lasagne) ist im Parameter **name** enthalten. Und der Parameter **essen** schließlich ist dann TRUE, wenn der Benutzer die Pizza bzw. Lasagne essen will und FALSE, wenn er sie bestellt. Wenn essen TRUE ist, zeigt Bestellen einen **CautionAlert**, sonst einen **StopAlert**, in dem dem Benutzer eine gute Mahlzeit gewünscht wird. Was der Benutzer genau essen will, wird vorher durch **ParamText** an den Alert übergeben. In beiden Alerts tauchen ja dafür die Platzhalter '^0' und '^1' auf.

Bestellen ist eine recht simple Prozedur, die im wesentlichen nur die Verwendung von Text-Parametern und von Alerts demonstrieren soll. Sie ist sehr programmspezifisch und für eine Weiterentwicklung des Rahmenprogramms von geringem Wert.

11.2.6 Änderungen in der Prozedur DoCommand

In **DoCommand** wurde bis jetzt der erste Befehl aus dem Apfel-Menü ignoriert. Nur die Auswahl des Namens einer Schreibtisch-Utensilie hatte eine Aktion zur Folge.

```
CONST
        idUeber      = 1002;

PROCEDURE UeberRahmen;
VAR
    dialog:     DialogPtr;
    dialog:     DialogPtr;
    dialog:     DialogPtr;
BEGIN
    dialog := GetNewDialog(idUeber,NIL,Pointer(-1));
    REPEAT
        ModalDialog(NIL,itemHit);
    UNTIL (itemHit=1); { OK-Knopf }
    DisposDialog(dialog);
END;{UeberRahmen}

PROCEDURE DoCommand(menuID,punkt: INTEGER);
BEGIN
    HiliteMenu(0);
    HiliteMenu(menuID);
    CASE menuID OF
        mApfel:
            BEGIN
            IF punkt = 1 THEN
                UeberRahmen             { NEU ! }
            ELSE
                BEGIN
                GetItem(
                    menu[mApfel],
                    punkt,
                    daName);
                OeffneDA(daName);
                END;
            END;{mApfel}
```

```
          ......
          { Der Rest bleibt unverändert }

      END; {CASE menuID}
   HiliteMenu(0);
END; {DoCommand}
```

Wählt der Benutzer den Befehl ***Über das Rahmenprogramm...*** aus dem Apfel-Menü aus, so wird in **DoCommand** die Prozedur **UeberRahmen** aufgerufen. **UeberRahmen** liest den Dialog mit dem Schlüssel ein, der in der globalen Konstante **idUeber** festgelegt wurde. Dieser Dialog wird dann mit **ModalDialog** behandelt, so lange, bis der Benutzer den OK-Knopf drückt. (Das genaue Aussehen dieses simplen Dialogs wurde bereits in einem vorangegangenen Abschnitt beschrieben.)

UeberRahmen ist simpel, aber ein gutes Beispiel für eine solche Prozedur. Alle "Gags" und netten Einfälle, die erscheinen sollen, wenn der Befehl ***Über das Rahmenprogramm...*** aufgerufen wird, können probemlos mit dem Resource-Editor in die entsprechende Resource eingebaut werden. Vom Programm aus ist selten mehr nötig, als hier gezeigt wurde.

11.2.7 Änderungen an SchliesseFenster

Da **BeendeProgramm** alle Fenster zu schließen versucht und **SchliesseFenster** bis jetzt nur mit Fenstern des Typs **grafKind** oder Fenstern von Schreibtisch-Utensilien fertig wird, muß auch hier eine kleine Änderung eingebaut werden, durch die **SchliesseFenster** auch fähig wird, Dialog-Fenster zu schließen.

```
PROCEDURE SchliesseFenster(fenster: WindowPtr);
BEGIN
    IF (GetWKind(fenster) = grafKind) THEN
        BEGIN
        { Bleibt unverändert }
        END
    ELSE IF (GetWKind(fenster) < 0) THEN
        BEGIN
        { Bleibt unverändert }
        END
    ELSE IF (GetWKind(fenster) = dialogKind) THEN
        DisposDialog(fenster);
END; {SchliesseFenster}
```

Die bewußte Änderung besteht nur aus einer zusätzlichen Abfrage auf das Feld **kind** des entsprechenden WindowRecords. Ist dieses gleich **dialogKind** (eine Konstante, die vom WindowManager definiert wird), schließt **SchliesseFenster** diesen Dialog mit **DisposDialog**. Diesem Aufruf liegt allerdings die Annahme zugrunde, daß alle Dialoge des Programms erzeugt wurden, ohne für sie vorher Platz zu reservieren – die Speicherplatzreservierung also dem DialogManager überlassen wurde.

11.2.8 Hilfsoperationen zur Behandlung von Schaltern

Um die Bearbeitung und das Abfragen von Schaltern zu vereinfachen, werden im Rahmenprogramm einige Hilfsoperationen implementiert. Diese beheben einige Schwächen der ToolBox, d.h., sie realisieren Funktionen, die eigentlich in die ToolBox gehört hätten.

```
FUNCTION  SchalterAn
            (dialog:   DialogPtr;
             schalter: INTEGER
            ): BOOLEAN;
VAR
    typ:      INTEGER;
    item:     Handle;
    box:      Rect;
BEGIN
    GetDItem(dialog,schalter,typ,item,box);
    IF (BitAnd(typ,3) = chkCtrl)
    OR (BitAnd(typ,3) = radCtrl) THEN
        SchalterAn :=
            (GetCtrlValue(ControlHandle(item)) = 1)
    ELSE
        SchalterAn := FALSE;
END; {SchalterAn}
```

SchalterAn testet zunächst, ob es sich bei der bewußten DialogKomponente überhaupt um einen Schalter handelt. Hierzu werden die beiden unteren Bits des Komponententyps mit den vordefinierten Konstanten **chkCtrl** und **radCtrl** verglichen. Durch die Beschränkung des Vergleichs auf die beiden unteren Bits spielen die additive Konstante **CtlItem** und ein eventuelles **disabled**-sein der entsprechenden Komponente keine Rolle.

Danach wird der Zustand des bewußten Schalters mit der Funktion **GetCtlValue** erfragt. Ergibt **GetCtlValue** 1, ist der Schalter "Ein". Ergibt sie 0, ist der Schalter aus.

Diese Realisierung von **SchalterAn** nutzt die Tatsache, daß eine Komponente, die als Schalter dargestellt wird, in Wirklichkeit eine *Kontrolle* ist. Deshalb auch die Umgehung des Pascal-Typschutzes beim Aufruf der Funktion **GetCtlValue**, die einen Parameter vom Typ ControlHandle erwartet. Kontrollen sind Objekte, die mit den Operationen des ControlManagers bearbeitet werden, der im Rahmen dieses Buches nicht weiter erläutert wird. Benutzt werden auch nur die beiden Operationen **GetCtlValue** und **SetCtlValue**, die den Wert, den eine Kontrolle besitzt, lesen bzw. verändern können.

```
PROCEDURE EinSchalter
            (dialog:   DialogPtr;
             schalter: INTEGER);
VAR
    typ:      INTEGER;
    item:     Handle;
    box:      Rect;
BEGIN
    GetDItem(dialog,schalter,typ,item,box);
    IF (BitAnd(typ,3) = chkCtrl)
    OR (BitAnd(typ,3) = radCtrl) THEN
        SetCtrlValue(ControlHandle(item,1);
END; {EinSchalter}
```

EinSchalter prüft zunächst, ob es sich bei der bewußten DialogKomponente überhaupt um einen Schalter handelt. Danach wird der Wert der Kontrolle, die der Schalter ja in Wirklichkeit ist, auf 1 gesetzt. Dies entspricht einem Schalter im Zustand "Ein". Die Prozedur **SetCtlValue** ist eine Operation des ControlManagers, der im Rahmen dieses Buches nicht weiter erläutert wird.

```
PROCEDURE AusSchalter
            (dialog:   DialogPtr;
             schalter: INTEGER);
VAR
    typ:      INTEGER;
    item:     Handle;
    box:      Rect;
BEGIN
    GetDItem(dialog,schalter,typ,item,box);
    IF (BitAnd(typ,3) = chkCtrl)
    OR (BitAnd(typ,3) = radCtrl) THEN
        SetCtrlValue(ControlHandle(item,0);
END; {AusSchalter}
```

AusSchalter prüft zunächst, ob es sich bei der bewußten DialogKomponente überhaupt um einen Schalter handelt. Danach wird der

Wert der Kontrolle, die der Schalter ja in Wirklichkeit ist, auf 0 gesetzt. Dies entspricht einem Schalter im Zustand "Aus".

```
PROCEDURE KippSchalter
            (dialog:   DialogPtr;
             schalter: INTEGER);
BEGIN
    IF SchalterAn(dialog,schalter) THEN
        AusSchalter(dialog,schalter)
    ELSE
        EinSchalter(dialog,schalter);
END; {KippSchalter}
```

KippSchalter ist die einfachste und nicht unbedingt eleganteste Lösung für das gestellte Problem. Die Prozedur fragt einfach ab, ob der bewußte Schalter sich im Zustand "Ein" befindet, und schaltet ihn in diesem Fall aus und ansonsten an.

```
TYPE
    SetOfSchalter= SET OF 1..256;

PROCEDURE RadioEin
            (dialog:   DialogPtr;
             schalter: INTEGER;
             gruppe:   SetOfSchalter);
VAR
    i:  INTEGER;
BEGIN
    FOR i := 1 TO 256 DO
        IF (i IN gruppe) THEN
            AusSchalter(dialog,i);
    EinSchalter(dialog,schalter);
END; {RadioEin}
```

RadioEin behebt einen — meiner Meinung nach — recht lästigen Mangel der ToolBox. Das Verhalten von Radio-Schaltern, die Gruppen bilden, von denen immer nur einer im Zustand "Ein" sein kann, muß vollständig vom Programm realisiert werden. Die ToolBox kennt keine Unterstützung für zusammengehörige Gruppen von Dialogkomponenten.

RadioEin speichert eine Grupppe von Dialogkomponenten (in diesem Falle Schaltern) in einer Pascal-Menge ab. Bevor einer der Schalter aus dieser Menge eingeschaltet wird, werden einfach vorher alle anderen explizit ausgeschaltet. Die Realisierung einer Schalter-Gruppe als Menge ist bestimmt keine ideale oder besonders elegante Lösung, stellt aber für die meisten Zwecke eine schnelle und ausreichende Lösung dar (allerdings nur in der Sprache Pascal!).

11.3 Schlußbemerkung

Dialoge sind ein sehr wichtiges Kommunikationsmittel des Macintosh mit seinem Benutzer. In Dialogen können recht komfortable Formulare realisiert werden, die nicht nur optisch ansprechend, sondern auch relativ leicht und fehlerfrei auszufüllen sind.

Der DialogManager verlangt allerdings im Gegensatz zu andern Paketen der ToolBox einiges Eingreifen von seiten des Programms — auch für einige recht simple Aktionen, wie z.B. den Wechsel eines Schalters vom Zustand "Ein" zum Zustand "Aus". Es ist deshalb auch recht schwierig, die Behandlung von Dialogen in allgemeingültiger Form zu programmieren. Im Rahmen diese Kapitels konnte deshalb nur die Behandlung der wichtigsten Standard-Komponenten von Dialogen erläutert werden, und auch dies in etwas lückenhafter Form.

Wer sich intensiver mit Dialogen beschäftigen und vor allem auch Möglich - keiten nutzen möchte, die über diese Standard-Komponenten hinausgehen, der sollte sich neben dem Kapitel *DialogManager* von *Inside Macintosh* vorher auch mit den Kapiteln *TextEdit* und *ControlManager* beschäftigen.

Anhang A: Übertragung in andere Sprachen

Wer dazu gezwungen ist, die Angaben und Beispiele aus *Inside Macintosh* und aus dem vorliegenden Buch in eine andere Sprache zu übertragen, wird auf einige Schwierigkeiten stoßen. Besonders *Inside Macintosh* geht stets von der Sprache LisaPascal aus, auf die sich alle Beschreibungen beziehen, sofern es sich nicht um Hinweise für Programmierer in Assembler-Sprache handelt. Deshalb sollen hier nun einige Hilfen folgen, die diese Übertragung etwas unproblematischer machen können. Es handelt sich im wesentlichen dabei aber nicht um fertige Rezepte, sondern eher um ein paar Tips, auf welche Stellen man achten sollte.

Besonders wichtig für eine problemlose Übertragung von Programmen ist auch die Kenntnis der Mechanismen, über die die ToolBox mit anderen Programmen kommuniziert. Zunächst etwas darüber, welche Annahmen die ToolBox über die Programme macht, von denen aus ihre Operationen aufgerufen werden.

Nahezu alle Teile der ToolBox gehen implizit davon aus, daß sie von einem Pascal-Programm oder von anderen Teilen der ToolBox aus aufgerufen werden. Einige ToolBox-Routinen können allerdings nur von Assembler aus aufgerufen werden, da sie einige ihrer Parameter in Registern der CPU erwarten. Die anderen Operationen der ToolBox erwarten ihre Parameter (und legen eventuelle Funktionsergebnisse) auf dem Stack der CPU und entsprechen damit den Konventionen, wie sie auch von den meisten Compilern beim Aufruf von Prozeduren angewandt werden. Es gibt dabei jedoch eine Menge feiner Unterschiede, da fast jeder Compiler-Schreiber von einer anderen Philosophie ausgeht.

Parameter-Übergabe

Der Aufruf einer Prozedur bzw. Funktion der ToolBox spielt sich normalerweise ab wie folgt. Handelt es sich um eine Funktion, wird auf dem Stack zunächst Platz für das Funktionsergebnis reserviert. Meist wird dafür eine 0 auf den Stapel gelegt. Dann werden die Parameter der Operation auf

den Stapel gelegt. Sie werden in der Reihenfolge auf den Stapel gelegt, wie sie im entsprechenden Prozedur-Kopf deklariert sind (in *Inside Macintosh* bzw. den verschiedenen Kapiteln dieses Buches nachzuschlagen).

Jeder Parameter und der reservierte Platz für das Funktionsergebnis nehmen dabei zwischen 2 und 4 Byte auf dem Stack ein. Die folgende Tabelle gibt an, wieviele Bytes für einen Parameter eines bestimmten Pascal-Typs auf den Stack gelegt werden:

BOOLEAN	2 Byte
CHAR	2 Byte
INTEGER	2 Byte
LONGINT	4 Byte
Alle Pointer	4 Byte

Alle VAR-Parameter werden als Pointer übergeben, benötigen also 4 Byte auf dem Stack. Zusammengesetzte Variablen, wie Arrays und Records werden ebenfalls als Pointer übergeben, sofern ihre Gesamtgröße 4 Byte übersteigt. Bleibt ihre Größe allerdings unter 4 Byte (z.B. Variablen vom Typ Point), werden sie als 2 oder 4 Byte große Werte auf den Stack gelegt.

Aufruf einer ToolBox-Routine

Nachdem der Aufruf durch das "Stapeln" der Parameter vorbereitet wurde, erfolgt der eigentliche Sprung in die ToolBox. Hierbei wird ein besonderer Trick angewandt.

Stößt die CPU des Macintosh auf einen Befehl, den sie nicht verarbeiten kann, so bricht sie das Programm ab und arbeitet ein ganz bestimmtes anderes Programm ab, das normalerweise versuchen sollte, diesen Fehler aufzufangen. Dies ist der sogenannte *Trap*-Mechanismus (engl. *trap* = Falle, Falltür). Er tritt nicht nur bei unbekannten Befehlen, sondern auch bei der Verarbeitung von Programm-Unterbrechungen (Interrupts) in Aktion. Es gibt eine ganze Reihe solcher Befehle, die die CPU nicht versteht, so z.B. alle, deren Hexadezimal-Schreibweise mit einem 'A' bzw. einem 'F' beginnt. Diese "falschen" Programm-Befehle werden von der ToolBox etwas "mißbraucht".

Jede Operation der ToolBox wird aufgerufen, indem ein solcher unbekannter Befehl (eine *Trap-Nummer*) in das Programm eingebaut wird. Das kleine Programm, das vom Trap-Mechanismus daraufhin aufgerufen wird, schaut dann nach, welche Zahl genau an der Stelle stand, an der das eigentliche

Programm abgebrochen wurde, und kann so feststellen, welcher Befehl der ToolBox aufgerufen wurde. Dieser wird dann ausgeführt. Nachdem die Abarbeitung des Befehls beendet ist, wird das eigentliche Programm hinter der Stelle, an der der "fehlerhafte" Befehl stand, fortgeführt. Auf diese Weise ist jede Operation der ToolBox eigentlich ein eigener Befehl der CPU MC68000.

Das Anspringen der Prozedur, die die ToolBox-Operation wirklich realisiert, geschieht dabei über eine sog. "Sprung-Tafel" oder "Sprung-Tabelle", die im Speicher (RAM) des Macintosh liegt. In dieser Tafel wird jeder Trap-Nummer eine Adresse im ROM (manchmal auch im RAM) zugeordnet, an der die Routine steht, die aufgerufen werden soll, wenn diese Trap-Nummer in einem Programm auftaucht. Durch die Sprung-Tabelle wird gewährleistet, daß alle Programme, die den Trap-Mechanismus nutzen, auch mit späteren Versionen des Macintosh (insbesondere des Macintosh-ROMs) kompatibel sind. Selbst wenn sich die Positionen der entsprechenden Routinen im ROM ändern, können die Trap-Nummern doch gleich bleiben. Auch mit der augenblicklichen Version des ROMs werden einige Fehler, die sich noch in den ROM-Routinen befinden, schon auf diesem Wege beseitigt. Beim Einschalten des Macintosh werden dazu einfach die Zeiger in der Sprungtafel "verbogen". Sie zeigen dann nicht mehr auf Routinen im ROM, sondern auf den Programm-Code, der ebenfalls beim Start in dem RAM geladen wurde.

Erzeugung von Trap-Nummern durch den Compiler

Wie bereits beschrieben wurde, geschieht der Einstieg in die ToolBox üblicherweise über den Trap-Mechanismus. Dazu erzeugt ein Compiler typischerweise Code, um alle Parameter auf den Stapel zu legen und eventuell Platz für das Funktionsergebnis zu reservieren, und legt dahinter eine Trap-Nummer, die von der CPU nicht als Befehl verstanden werden kann.

Nicht jeder Compiler ist zur Erzeugung solchen Codes fähig. Manchmal sind Prozeduren für einen Compiler nur Adressen, und wenn eine Prozedur aufgerufen werden soll, so erzeugt er immer den folgenden Code:

```
JSR ProcAdresse
```

Wobei **JSR** den Sprungbefehl zu einer Prozedur darstellt. Da ToolBox-Prozeduren aber so nicht angesprungen werden können, müssen in diesem Fall kleine Assembler-Routinen geschrieben werden, die dies ermöglichen. Der Compiler erzeugt dann zunächst einen Sprungbefehl zur Assembler-Routine, die in ihrem Code dann erst die eigentliche Trap-Nummer der

ToolBox-Prozedur enthält. Diese Assembler-Routinen werden dann in Programm-Bibliotheken gelegt, mit denen jedes Programm vor der Ausführung verbunden (ge*linkt*) werden muß.

Diese Lösung ist ein wenig unglücklich, da jedes Programm einen kompletten Satz dieser kleinen Assembler-Routinen (für jede ToolBox-Routine eine) mit sich führen muß. Besser sind Compiler, die im eigentlichen Programm-Code bereits Trap-Nummern erzeugen können.

Problemgebiete bei einer Übertragung in andere Sprachen

Möchte man ein Programm-Beispiel aus diesem Buch oder aus *Inside Macintosh* in eine andere Sprache als LisaPascal übertragen, so tauchen dabei typischerweise die Probleme auf, die im folgenden kurz beschrieben werden sollen.

- Übergabe von Parametern an ToolBox-Prozeduren; dabei insbesondere:
  - Reihenfolge
  - VAR-Parameter
  - **Point**-Parameter
  - **ResType**-Parameter
  - **BOOLEAN**-Parameter und Funktionsergebnisse
- Strings
- Zeiger auf Prozeduren

Probleme mit der Parameter-Reihenfolge

Wenn ein neuer Compiler für den Macintosh auftaucht, so existierte dieser meist vorher schon in anderen Versionen für andere Rechner. Es handelt sich also üblicherweise nicht um neu geschriebene Compiler, sondern um Übertragungen schon existierender. Deshalb ist es oft der Fall, daß diese Compiler andere Konventionen für die Übergabe von Parametern an Prozeduren haben, als dies die ToolBox (LisaPascal) erwartet.

Ein (kleines) Problem ist manchmal die Reihenfolge der Parameter. Die Programmiersprache C legt z.B. aus verschiedenen Gründen die Parameter für eine Prozedur in der umgekehrten Reihenfolge ihrer Deklaration auf den Stack. Also zuerst der zuletzt deklarierte und darüber dann die vorangehenden Parameter. Zudem tendieren gute C-Compiler dazu, gelegentlich auch Parameter in Registern zu übergeben und Funktionsergebnisse in Registern zu empfangen.

Diese Probleme werden allerdings meist schon von den Compiler-Schreibern gelöst, indem sie den Aufruf von ToolBox-Routinen wieder über Assembler-Prozeduren regeln, in denen die Reihenfolge der Parameter umgekehrt wird. Dies ist aber langsam, umständlich und fehleranfällig, da diese Assembler-Routinen praktisch für jede ToolBox-Operation gesondert geschrieben werden müssen. Andere Lösungen sind z.B. Direktiven, die im Programm stehen müssen und dem Compiler explizit sagen, daß es sich bei der folgenden Prozedur nicht um eine "normale" handelt, sondern bei ihrem Aufruf spezielle Pascal-Regeln angewandt werden müssen.

Doch nicht alle Probleme können so vom Compiler bzw. den zugehörigen Bibliotheken beseitigt werden. In einigen Programmiersprachen (z.B. C und Modula-2) ist es möglich, Variablen zu definieren, die auf Prozeduren und Funktionen zeigen. Diese werden zwar nicht allzuhäufig benötigt, können aber zu Schwierigkeiten führen, wenn man versucht, in ihnen Zeiger auf ToolBox-Prozeduren zu speichern. So etwas wird nahezu in allen Fällen zu katastrophalen Fehlern führen!

Probleme mit Zeigern auf Parameter

Ein viel häufiger auftauchendes Problem als das der Parameter-Reihenfolge sind die VAR-Parameter von ToolBox-Prozeduren und Parameter, die größer sind als 4 Byte. Solche Parameter werden von LisaPascal, wie oben beschrieben, nie selbst auf den Stack gelegt, sondern stets nur ein 4 Byte großer Zeiger auf die wirklichen Daten.

Der LisaPascal-Compiler tut dies ganz automatisch; der Programmierer merkt nichts davon – ja, es kann dem Prozeduraufruf nicht einmal angesehen werden, welcher Parameter als Adresse und welcher als Wert übergeben wird. In fast allen anderen Programmiersprachen ist es aber nötig, die Adresse einer Variablen explizit zu ermitteln und an eine Prozedur zu übergeben, wenn diese eine Adresse erwartet.

Bevor man eine solche Prozedur aufruft, sollte man sich ihre Definition also ganz genau anschauen. Hat sie VAR-Parameter oder Parameter, die größer sind als 4 Byte muß deren Adresse übergeben werden. In MacAdvantage UCSD-Pascal sieht das z.B. aus wie folgt:

```
procedure PaintRect(r : Rect);
......
PaintRect(locate(einRechteck));
```

wobei die eingebaute Funktion **locate** die Adressenbestimmung erledigt. In C sähe das entsprechende Programmstück aus wie folgt:

```
PaintRect(&einRechteck);
```

wobei der Operator **&** die Adresse der Variablen **einRechteck** bestimmt. Besonders problematisch sind in C die VAR-Parameter. Da C ja keinen Typ-Schutz und keine Überprüfung der Parameterlisten von Funktions-Aufrufen kennt, ist es oft problemlos möglich, statt der Adresse einer Variablen deren aktuellen Wert zu übergeben — ein unter Umständen schwerwiegender Fehler mit katastrophalen Folgen.

Strukturierte Werte mit vier oder weniger Byte Größe

Ein anderes Problem im Zusammenhang mit der Übergabe von Parametern sind sehr kleine strukturierte Werte. Variablen vom Typ **Point** (zwei INTEGER) oder **ResType** (vier CHARs) bleiben gerade unterhalb der Grenze von 4 Byte, oberhalb der Variablen nur noch in Form von Zeigern übergeben werden. Sowohl Punkte wie auch die vierbuchstabigen Kennungen von Resourcen müssen aber als ein zusammenhängender Block von 4 Byte auf den Stack gelegt werden, wenn eine Prozedur der ToolBox einen solchen Parameter erwartet.

Einige Compiler weigern sich aber, strukturierte Werte (RECORDs und ARRAYs) als Werte auf den Stack zu legen. Sie erwarten bei strukturierten Werten immer die entsprechende Adresse, egal, wieviel Platz der Wert benötigt. Hierzu gibt es einige Tricks, die meist das Konzept der *Varianten Records* (in Pascal) oder der *Union* (in C) nutzen. Diese Tricks sind aber von Compiler zu Compiler sehr unterschiedlich und sollen deshalb hier nicht näher beschrieben werden. Der Leser sollte aber in der Dokumentation seiner Sprache danach Ausschau halten und gegebenenfalls immer auf der Hut sein, wenn er strukturierte Datentypen verwendet, deren Platzbedarf geringer als oder gleich 4 Byte ist.

Parameter vom Typ BOOLEAN

Variablen des Typs BOOLEAN (ein Wahrheitswert: WAHR oder FALSCH bzw. TRUE oder FALSE) benötigen eigentlich nur ein Bit zur Speicherung. LisaPascal (und somit die ToolBox) verwendet meist ein Byte oder zwei Byte für ihre Darstellung. Dies ist in anderen Sprachen und bei anderen Compilern durchaus ähnlich, aber eben nicht genau gleich!

Wird ein BOOLEAN-Wert z.B. innerhalb eines Records abgespeichert und direkt dahinter folgt im selben Record ein Wert, der nur ein Byte benötigt, so legt LisaPascal auch den BOOLEAN-Wert in einem Byte ab. Das niederwertigste Bit dieses Records enthält dann den eigentlichen Wert. Ist es 1, so ist der Wert dieses Record-Feldes TRUE; ist es 0, so ist der Wert FALSE. Folgt jedoch auf das BOOLEAN-Feld ein anderes Feld, das 2 oder mehr Byte benötigt, oder handelt es sich um eine einzelne BOOLEAN-Variable oder einen Parameter vom Typ BOOLEAN, so werden dafür 2 Byte (16 Bit) reserviert. Das entscheidende Bit dieser 16 Bit ist nun nicht etwa das niederwertigste, sondern das Bit mit der Nummer 8 (wobei das niederwertigste Bit als Bit 0 gilt) — also das niederwertigste Bit des höherwertigen Bytes.

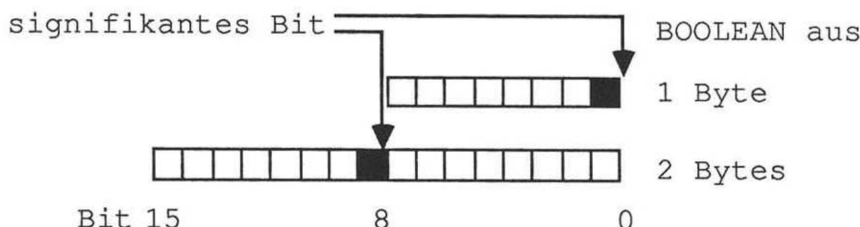


Abb. A - 1: *Speicherung BOOLEscher Werte*

Nicht alle Compiler kommen mit solch komplizierten Packungsregeln zu Rand und deshalb ist gerade die Verwendung BOOLEscher Werte in Records, als Prozedur-Parameter und als Funktions-Ergebnis etwas problembehaftet. Manche Compiler kennen bis zu drei verschiedene Datentypen, die dem einen Datentyp BOOLEAN von LisaPascal entsprechen. Je nachdem, wo ein BOOLEscher Wert auftaucht, wird der eine oder der andere Typ verwendet.

Falls möglich, sollte der Leser immer versuchen, BOOLEsche Werte als Zahlen aufzufassen. Ist der Wert Null , so ist die BOOLEsche Entsprechung dazu FALSE, ist der Wert ungleich Null, so bedeutet das TRUE. Man sollte bei einer Abfrage eines solchen Wertes aber nie mit einem bestimmten Wert für TRUE vergleichen, die Abfrage auf *ungleich Null* (<> 0) ist wesentlich zuverlässiger. Je nachdem, wo der Wert dann auftaucht, handelt es sich dann um eine 16-Bit-Zahl (**INTEGER** in Pascal und Modula; **int** in C) oder um eine 8-Bit-Zahl (**Byte** (= -128..127) in Pascal und Modula; **char** in C).

Am problematischsten sind BOOLEsche Werte in Sprachen, die selbst einen BOOLEschen Datentyp kennen, wie Pascal und Modula-2. Wenn deren eigene Speicherungsmethodik von der von LisaPascal abweicht, kann deren Datentyp BOOLEAN nicht für die BOOLEschen Werte der ToolBox genutzt werden, was die Programme nicht gerade leserlicher macht. Sprachen, die keinen BOOLEschen Datentyp kennen, wie z.B. C, haben es da etwas leichter, da in ihnen logische Entscheidungen zumeist sowieso schon aufgrund eines numerischen Wertes getroffen werden (meist aufgrund des Vergleichs *gleich Null* oder *ungleich Null*).

Strings

Besonders schwierig gestaltet sich im allgemeinen auch die Übergabe von Strings an die ToolBox und besonders die Verwendung von Strings, die man sowohl an die ToolBox übergeben will als auch selbst im Programm bearbeiten will. In einem gut geschriebenen Programm sollten solche Fälle zwar selten sein, da in diesen alle Strings im Bedarfsfall aus der Resource-Gabel eingelesen und direkt an die entsprechenden ToolBox-Operationen weitergegeben werden, machmal kommt man aber nicht ohne Strings im Programm aus.

Fast jede Sprache hat nun ihre eigene Methode, Strings abzuspeichern. In Pascal wird ein String im allgemeinen (einen Standard hierfür gibt es allerdings nicht) als Folge seiner Buchstaben abgespeichert, also als ARRAY...OF CHAR, der ein Byte (Buchstabe) vorangeht, in dem die Länge des folgenden Strings steht. Dies begrenzt die Länge von Strings somit auf eine maximale Länge von 255 Buchstaben, denn größere Zahlen als 255 kann man in das eine Längen-Byte ja nicht eintragen. In C und Modula werden die Buchstaben des Strings einfach hintereinandergeschrieben — bereits der erste Buchstabe ist also signifikant — und das Ende des Strings wird durch einen bestimmten Buchstaben markiert. Dies ist in C und Modula-2 natürlich nicht derselbe Buchstabe.

Würde man jetzt in C oder Modula-2 einen String einer Variablen zuweisen und diese Variable z.B. in einem **DrawString**-Aufruf verwenden, so würde die ToolBox in dem ersten Buchstaben des Strings eine Längenangabe sehen und deshalb entweder nicht den ganzen Text zeichnen oder einen längeren Text, der dann am Ende wahrscheinlich Quatsch enthält.

Umgekehrt wird ein String, den man von der ToolBox erhält, z.B. über einen **GetString**-Aufruf aus der Resource-Gabel, von einem Modula-2- oder C-Programm nicht richtig verarbeitet werden können. Verwendet man diesen

String z.B. mit einer Modula-2-Prozedur, die zwei Strings hintereinanderhängt, so wird diese Prozedur nach dem Begrenzer-Zeichen suchen, das das String-Ende markiert. Diese wird sie manchmal zu früh und meistens wohl zu spät finden, da diese Begrenzer-Zeichen in normalen Strings nie oder nur sehr selten vorkommen.

In den meisten Programmiersprachen gibt es deshalb Konvertierungs-Prozeduren, die zwischen der Darstellung von Strings in der jeweiligen Sprache und der der ToolBox (Pascal) hin und her wechseln können. Dies ist umständlich, aber erträglich, da, wie bereits oben gesagt, die String-Verarbeitung in einem guten Macintosh-Programm eigentlich selten sein sollte.

Manchmal wird aber auch eine automatische Konvertierung bei der Übergabe von String-Parametern gemacht. Solche Parameter werden vor der Übergabe an die Prozedur, ohne daß der Programmierer es merkt, in das ToolBox-Format umgewandelt. Diese Lösung scheint mir aber um einiges fehlerträchtiger, da dem Programmierer die Kontrolle über das String-Format genommen wird. Fehler, die mit einer solchen automatischen Konvertierung zusammenhängen, können sehr schwer zu finden sein.

Die Fehler, die auftreten können, sind prinzipiell aber bei beiden Methoden der Konvertierung (automatisch und manuell) dieselben. Entweder unterbleibt die Konvertierung, wo sie eigentlich nötig gewesen wäre, oder sie wird zweimal oder häufiger hintereinander in derselben Richtung gemacht. Wie man sich leicht vorstellen kann, kann ja eigentlich nur Quatsch herauskommen, wenn ein String zweimal nacheinander z.B. vom Pascal-Format ins C-Format umgewandelt wird. Nach der ersten Umwandlung steht ja an erster Stelle überhaupt kein Längen-Byte mehr.

Wann immer der Leser bei einem seiner eigenen Programme merkwürdige Texte auf dem Bildschirm findet, bei denen entweder an erster Stelle ein merkwürdiger Buchstabe steht oder die letzten Buchstaben völligen Quatsch enthalten, sollte er nach solchen Fehlern im Programm suchen!

Zeiger auf Prozeduren

Einige ToolBox-Operationen verlangen Zeiger auf Prozeduren bzw. Funktionen als Parameter. Diese Zeiger werden in Datenstrukturen der ToolBox gespeichert und die entsprechenden Prozeduren bei Bedarf aufgerufen. Aus C oder Modula-2 sind ähnliche Konstruktionen bekannt; in Pascal ist so etwas eigentlich unmöglich. Benötigt werden diese Zeiger auf

Prozeduren z.B. für modale Dialoge und Alerts, die im Kapitel 11 erläutert wurden. In allen Beispielen wurde allerdings immer NIL für solche Parameter übergeben.

Wenn die ToolBox eine solche Prozedur oder Funktion aufruft, geht sie dabei immer von der Annahme aus, daß es sich dabei um eine Prozedur handelt, die sich verhält, als wäre sie in LisaPascal geschrieben. Dies betrifft vor allem die Lage der Parameter auf dem Stack, die Position eines eventuellen Funktionsergebnisses usw. Programmiert man aber in einer anderen Sprache, so ist dies normalerweise nicht gewährleistet. Selbst wenn es also möglich wäre, einen Zeiger auf eine Prozedur zu bestimmen — fast alle Programmiersprachen für den Macintosh sehen solche Möglichkeiten vor — so würde doch der Aufruf dieser Prozedur durch die ToolBox normalerweise schiefgehen.

In verschiedenen Sprachen gibt es deshalb verschiedene Konstruktionen, mit diesem Problem fertig zu werden. Manche C-Compiler besitzen z.B die Möglichkeit, mit einer bestimmten Compiler-Direktive auch C-Funktionen zu schreiben, die sich verhalten, als wären sie in LisaPascal geschrieben. Hierfür muß normalerweise das Wort PASCAL vor den Funktions-Kopf geschrieben werden. Nur Zeiger auf solche Prozeduren sollten an die ToolBox übergeben werden. In der Sprache C, die ja auch die Möglichkeit kennt, Zeiger auf Prozeduren zu speichern und sie über diese Zeiger aufzurufen, sollte mit Zeigern auf solche PASCAL-Prozeduren sehr vorsichtig umgegangen werden. Sie sollten nie zum Aufruf der entsprechenden Prozeduren von C-Funktionen aus verwendet werden, sondern immer nur für die Übergabe an die ToolBox.

Die Sprache MacAdvantage UCSD-Pascal kennt eine ganz andere, recht umständliche Konstruktion, Zeiger auf Prozeduren zu gewinnen, und mindestens ein Modula-Compiler erlaubt es überhaupt nicht, Zeiger auf Modula-Prozeduren an die ToolBox zu übergeben. In anderen Sprachen muß man wiederum auf Assembler zurückgreifen, wenn man diese Möglichkeit nutzen will.

Auf jeden Fall sollte man sehr vorsichtig sein, bevor man diese recht fortgeschrittene Möglichkeit der ToolBox nutzt. In manchen Sprachen ist es relativ problemlos und ungefährlich möglich, Zeiger auf Prozeduren zu verwenden; in anderen sollte man nur mit größtmöglicher Vorsicht vorgehen. Fehler, die auf falscher Verwendung von Prozedur-Zeigern beruhen, können sehr schwer zu finden sein.

Anhang B: Programmlisting

Im folgenden ist das Rahmenprogramm in seiner letzten Ausbaustufe noch einmal komplett aufgelistet. Es wird unkommentiert gelistet, da ausreichende Erläuterungen bereits in den vorangegangenen Kapiteln gegeben wurden. Dieses Programm kann so, wie vorgestellt, von einem LisaPascal-Compiler oder einem kompatiblen System übersetzt werden. Für andere Compiler sind mehr oder weniger große Umstellungen vorzunehmen. (Für die Programmiersprachen TML-Pascal und ON-STAGE-Pascal, die beide auf einem Macintosh mit mindestens 512 KByte Hauptspeicher laufen, dürften diese Änderungen aber minimal sein.)

```
PROGRAM Rahmen;
USES
      { Hier müßte eine Systemabhängige Liste von
        Bibliotheks-Units folgen, die die ToolBox-
        Zugriffe ermöglichen !
      }

CONST
  mApfel              = 1000;
      mUeber          = 1;

  mAblage             = 1001;
      mOeffne1        = 1;
      mOeffne2        = 2;
      mBeenden        = 4;

  mBearbeiten     = 1002;
      mUndo           = 1;

      mCut            = 3;
      mCopy           = 4;
      mPaste          = 5;
      mClear          = 6;
```

```
      mGroesser          = 8;
      mKleiner           = 9;
      mZeige1            = 11;
      mZeige2            = 12;

   firstMenu             = 1000;
   lastMenu              = 1002;

   grafKind              = 8;
   textKind              = 9;

   idWin1                = 1001;
   idWin2                = 1002;

   idStr                 = 1001;

   MaxSchalter           = 256;

TYPE
      SetSchalter = SET OF 1..MaxSchalter;

VAR
      ende:    BOOLEAN;
      menu:    ARRAY[firstMenu..lastMenu]OF MenuHandle;

      GrowRect:          Rect;
      DragRect:          Rect;

      fenster1,
      fenster2:          WindowPtr;

      nonModal:          DialogPtr;

      wPict:             PicHandle;
      scale:             INTEGER;

      fMouse,
      fShift,
      fShLock,
      fOption,
      fCommand,
      fActive:           BOOLEAN;

      dasEreignis:       EventRecord;
```

```
(* ********************************************
*********************************************

     Utility-Funktionen für die FENSTER-Verwaltung

*********************************************
****************************************** *)


FUNCTION  Sichtbar(fenster: WindowPtr): BOOLEAN;
BEGIN
     IF fenster <> NIL THEN
          Sichtbar := WindowPeek(fenster)^.visible;
     ELSE
          Sichtbar := FALSE;
END;{Sichtbar}



PROCEDURE SetWKind(fenster: WindowPtr;
                            kind: INTEGER);
BEGIN
     IF fenster <> NIL THEN
          WindowPeek(fenster)^.windowKind := kind;
END;{SetWKind}



FUNCTION GetWKind(fenster: WindowPtr): INTEGER;
BEGIN
     IF fenster <> NIL THEN
          GetWKind := WindowPeek(fenster)^.windowKind
     ELSE
          GetWKind := 0;
END;{GetWKind}
```

```
PROCEDURE InvalGrow(fenster: WindowPtr);
VAR
      savePort: GrafPtr;
      r:                 Rect;
BEGIN
      GetPort(savePort);

      SetPort(fenster);
      r := WindowPeek(fenster)^.port.portRect;
      r.left := r.right-15;
      InvalRect(r);
      r := WindowPeek(fenster)^.port.portRect;
      r.top := r.bottom-15;
      InvalRect(WindowPeek(fenster)^.port.portRect);

      SetPort(savePort);
 END;{InvalGrow}


PROCEDURE InvalFenster(fenster: WindowPtr);
VAR
      savePort:          GrafPtr;
      r:                 Rect;
BEGIN
      GetPort(savePort);

      SetPort(fenster);
      r := WindowPeek(fenster)^.port.portRect;
      with r do
            begin
            bottom := bottom-15;
            right  := right-15;
            end;
      InvalRect(r);

      SetPort(savePort);
END;{InvalFenster}
```

```
PROCEDURE WindowGrow(fenster: WindowPtr;
                pt: Point; growRect: Rect);
VAR
    growErg:            LONGINT;
BEGIN
    IF (GetWKind(fenster) = grafKind) THEN
        BEGIN
        growErg :=
            GrowWindow(fenster,pt.param,growRect);
        InvalGrow(fenster);
        SizeWindow(fenster,
                    LoWord(growErg),
                    HiWord(growErg),TRUE);
        InvalGrow(fenster);
        END;
END; {WindowGrow}


(* ********************************************
*********************************************

        Utility-Funktionen für die DIALOG-Verwaltung

*********************************************
******************************************* *)


PROCEDURE EinSchalter(dialog: DialogPtr;
                        derSchalter: INTEGER);
VAR
    itTyp:      INTEGER;
    item:       Handle;
    box:        Rect;
    name:       Str255;
    val:        INTEGER;

BEGIN
    GetDItem(dialog,derSchalter,itTyp,item,box);
    SetCtlValue(item,1);
END;{EinSchalter}
```

```
PROCEDURE AusSchalter(dialog: DialogPtr;
                              derSchalter: INTEGER);
VAR
      itTyp:        INTEGER;
      item:         Handle;
      box:          Rect;
      name:         Str255;
      val:          INTEGER;

BEGIN
      GetDItem(dialog,derSchalter,itTyp,item,box);
      SetCtlValue(item,0);
END;{AusSchalter}

PROCEDURE RadioEin
                  (dialog:  DialogPtr;
                  anSchalter: INTEGER;
                  gruppe:   SetSchalter);
VAR
      i:    INTEGER;
BEGIN
      FOR i := 1 TO MaxSchalter DO
            IF i IN gruppe THEN
                  AusSchalter(dialog,i);
      EinSchalter(dialog,anSchalter);
END;{RadioEin}

FUNCTION  Schalter(dialog: DialogPtr;
                              derSchalter: INTEGER): BOOLEAN;
VAR
      itTyp:        INTEGER;
      item:         Handle;
      box:          Rect;
      name:         Str255;
      val:          INTEGER;

BEGIN
      GetDItem(dialog,derSchalter,itTyp,item,box);
      Schalter := ( GetCtlValue(item) = 1 );
END;{Schalter}


PROCEDURE KippSchalter(dialog:  DialogPtr;
                              derSchalter: INTEGER);
BEGIN
      IF Schalter(dialog,derSchalter) THEN
            AusSchalter(dialog,derSchalter)
      ELSE
            EinSchalter(dialog,derSchalter);
END;{RadioEin}
```

```
(* ******************************************
*********************************************

       Haupt-Funktionen der FENSTER-Verwaltung

*********************************************
****************************************** *)

PROCEDURE SchliesseFenster(fenster: WindowPtr);
BEGIN
     IF (GetWKind(fenster) = grafKind) THEN
          BEGIN
          HideWindow(fenster);
          IF (fenster = fenster1) THEN
               BEGIN
               EnableItem(menu[mAblage],mOeffne1);
               DisableItem(menu[mBearbeiten],mZeige1);
               END
          ELSE IF (fenster = fenster2) THEN
               BEGIN
               EnableItem(menu[mAblage],mOeffne2);
               DisableItem(menu[mBearbeiten],mZeige2);
               END;
          END
     ELSE IF GetWKind(fenster) < 0 THEN
               CloseDeskAcc(GetWKind(fenster))
     ELSE IF GetWKind(fenster) = dialogKind THEN
               DisposDialog(fenster);
END;{SchliesseFenster}



PROCEDURE AenderFenster(fenster: WindowPtr);
VAR
     r:                 Rect;
     savePort:          GrafPtr;
BEGIN
     GetPort(savePort);

     SetPort(fenster);
     IF (GetWKind(fenster) = grafKind) THEN
          BEGIN
          WITH WindowPeek(fenster)^.port.portRect DO
               SetRect(r,left,top,right-15,bottom-15);
          ClipRect(r);
          END;

     SetPort(savePort);
END; {AenderFenster}
```

```
PROCEDURE ZeichneBild(bild: PicHandle;
                            wo: Point; scale: INTEGER);
VAR
      r:    Rect;
BEGIN
      r := bild^^.picFrame;
      OffsetRect(r,r.left,r.top);
      WITH r DO
            BEGIN
            OffsetRect(r,left,top);
            bottom := (bottom * scale) DIV 8;
            right  := (right  * scale) DIV 8;
            OffsetRect(r,wo.h,wo.v-bottom);
      END;
      DrawPicture(bild,r);
END;{ZeichneBild}


PROCEDURE ZeichneScroll(fenster:  WindowPtr);
VAR
      r:          Rect;
      clip:       Handle;
BEGIN

      PenNormal;
      clip := NewRgn;
      GetClip(clip);
      ClipRect(WindowPeek(fenster)^.port.portRect);

      WITH WindowPeek(fenster)^.port.portRect DO
            SetRect(r,right-15,top,right,bottom);
      FillRect(r,white);

      WITH WindowPeek(fenster)^.port.portRect DO
            SetRect(r,left,bottom-15,right-15,bottom);
      FillRect(r,white);

      DrawGrowIcon(fenster);

      SetClip(clip);
      DisposeRgn(clip);

END;{ZeichneScroll}
```

```
PROCEDURE ZeichneInhalt(fenster: WindowPtr);
VAR
     bei:         Point;
     wr:          WindowRecord;
BEGIN

     FillRect(WindowPeek(fenster)^.port.portRect,
                        white);
     LONGINT(bei) := GetWRefCon(fenster);
     ZeichneBild(wPict,bei,scale);

END;{ZeichneInhalt}



PROCEDURE KlickInFenster(fenster: WindowPtr;
                                   wo: Point);
VAR
     punkt:            Point;
BEGIN
     IF (GetWKind(fenster) = grafKind) THEN
          BEGIN
          LONGINT(punkt) := GetWRefCon(fenster);
          ZeichneBild(wPict,punkt,scale);
          ZeichneBild(wPict,wo,scale);
          WHILE StillDown DO
               BEGIN
               ZeichneBild(wPict,wo,scale);
               GetMouse(wo);
               ZeichneBild(wPict,wo,scale);
               END;
          SetWRefCon(fenster,LONGINT(punkt));
          END;
END; {KlickInFenster}
```

```
PROCEDURE WindowUpdate(fenster:  WindowPtr);
VAR
      savePort:         GrafPtr;
BEGIN
      GetPort(savePort);
      SetPort(fenster);
      BeginUpdate(fenster);
            IF (GetWKind(fenster) = grafKind) THEN
                  BEGIN
                  ZeichneInhalt(fenster);
                  ZeichneScroll(fenster);
                  END;
      EndUpdate(fenster);
      SetPort(savePort);
END;{WindowUpdate}


PROCEDURE WindowActivate(fenster: WindowPtr;
                                      aktiv: BOOLEAN);
BEGIN
      IF aktiv THEN
            SetPort(fenster);

      IF (GetWKind (fenster) = grafKind) THEN
            InvalGrow(fenster);

      IF (GetWKind (fenster) = grafKind)
      AND aktiv THEN
            BEGIN
            IF fenster = fenster1 THEN
                  BEGIN
                  DisableItem(menu[mAblage],mOeffne1);
                  DisableItem(menu[mBearbeiten],mZeige1);
                  IF Sichtbar(fenster2) THEN
                        EnableItem(menu[mBearbeiten],
                                          mZeige2);
                  END
            ELSE IF (fenster = fenster2) THEN
                  BEGIN
                  DisableItem(menu[mAblage],mOeffne2);
                  DisableItem(menu[mBearbeiten],mZeige2);
                  IF Sichtbar(fenster1) THEN
                        EnableItem(menu[mBearbeiten],
                                          mZeige1);
                  END;
            END;
END;{WindowActivate}
```

```
(* ******************************************
*******************************************

                              DIALOGE

*******************************************
***************************************** *)


PROCEDURE UeberRahmen;
VAR
     dialog:    DialogPtr;
     itemHit: INTEGER;

BEGIN
     dialog := GetNewDialog(1002,NIL,POINTER(-1));
     REPEAT
          ModalDialog(NIL,itemHit);
     UNTIL (itemHit = OK);
     DisposDialog(dialog);
END;{UeberRahmen}

PROCEDURE Bestellen(name:   Str255;
                            pizza:  BOOLEAN;
                            essen:  BOOLEAN);
VAR
     itemHit: INTEGER;
BEGIN
     IF pizza THEN
          ParamText(name,GetString(1001)^^,'','')
     ELSE
          ParamText(name,GetString(1002^^,'','');

     IF essen THEN
          itemHit := StopAlert(1004,NIL)
     ELSE
          itemHit := CautionAlert(1003,NIL);

END;{Bestellen}
```

```
PROCEDURE DialogKlick(dialog: DialogPtr;
                              itemHit: INTEGER);
CONST
      itPizza    = 4;
      itLasag    = 5;
      itTomat    = 6;
      itKase     = 7;
      itThunf    = 8;
      itName     = 9;
VAR
      itTyp:     INTEGER;
      item:      Handle;
      box:       Rect;
      name:      Str255;

BEGIN
      GetDItem(dialog,itemHit,itTyp,item,box);
      CASE itemHit OF
            1,2:
                  BEGIN
                  GetDItem(dialog,itName,itTyp,item,box);
                  GetIText(item,name);
                  Bestellen(name,
                              Schalter(dialog,itPizza),
                              (itemHit = 2));
                  END;
            itPizza:
                  RadioEin(dialog,itPizza,
                              [itPizza,itLasag]);
            itLasag:
                  RadioEin(dialog,itLasag,
                              [itPizza,itLasag]);
            itTomat,
            itKase,
            itThunf:
                  KippSchalter(dialog,itemHit);
      END{CASE};
END;{DialogKlick}
```

```
(* ******************************************
********************************************

                          DOCOMMAND

********************************************
****************************************** *)


PROCEDURE DoCommand(menuID,punkt: INTEGER);
VAR
     daName:     Str255;
     ref:        INTEGER;
     drvr:       OSType;
     res,
     dummy:      Handle;

BEGIN
     HiliteMenu(0);
     HiliteMenu(menuID);
     CASE menuID OF
          mApfel:
               BEGIN
               IF (punkt = mUeber) THEN
                    UeberRahmen
               ELSE
                    BEGIN
                    GetItem(menu[menuID],punkt,daName);
                    SetResLoad(FALSE);
                    res := GetNamedResource ('DRVR',
                                                   daName);
                    dummy := NewHandle(
                                  SizeResource(res)+3072);
                    ref   := MemError;
                    DisposHandle(dummy);
                    SetResLoad(TRUE);
                    IF (ref=0) THEN
                         ref := OpenDeskAcc(daName);
                    END;
               END;{mApfel}
          mAblage:
               BEGIN
               CASE punkt OF
                    mOeffne1:
                         BEGIN
                         ShowWindow(fenster1);
                         SelectWindow(fenster1);
                         END;
                    mOeffne2:
                         BEGIN
                         ShowWindow(fenster2);
```

```
                    SelectWindow(fenster2);
                    END;
                mBeenden:
                    ende := TRUE;
                END;{CASE punkt}
            END;{mAblage}
        mBearbeiten:
            BEGIN
            CASE punkt OF
                mUndo:
                    BEGIN
                    IF NOT SystemEdit(punkt-1) THEN
                        { passiert gar nichts };
                    END;{SystemEdit}
                mCut:
                    BEGIN
                    IF NOT SystemEdit(punkt-1) THEN
                        { passiert gar nichts };
                    END;
                mCopy:
                    BEGIN
                    IF NOT SystemEdit(punkt-1) THEN
                        { passiert gar nichts };
                    END;
                mPaste:
                    BEGIN
                    IF NOT SystemEdit(punkt-1) THEN
                        { passiert gar nichts };
                    END;
                mClear:
                    BEGIN
                    IF NOT SystemEdit(punkt-1) THEN
                        { passiert gar nichts };
                    END;
                mGroesser:
                    BEGIN
                    scale := scale * 2;
                    EnableItem(menu[mBearbeiten],
                                               mKleiner);
                    IF scale >= 64 THEN
                        BEGIN
                        scale := 64;
                        DisableItem(
                                menu[mBearbeiten],
                                       mGroesser);
                        END;
                    InvalFenster(fenster1);
                    InvalFenster(fenster2);
                    END;

                mKleiner:
```

```
                        BEGIN
                        scale := scale DIV 2;
                              EnableItem(menu[mBearbeiten,
                                             mGroesser);
                        IF scale <= 1 THEN
                              BEGIN
                              scale := 1;
                              DisableItem(menu[mBearbeite,
                                             mKleiner);
                              END;
                        InvalFenster(fenster1);
                        InvalFenster(fenster2);
                        END;
                  mZeige1:
                        SelectWindow(fenster1);
                  mZeige2:
                        SelectWindow(fenster2);
                  END;{CASE punkt}
            END;{mBearbeiten}
      END; {CASE menuID}
   HiliteMenu(0);
END; {DoCommand}


(* ********************************************
*********************************************

                        Event-Bearbeitung

*********************************************
****************************************** *)


PROCEDURE Mausklick(wo:  Point);
VAR
      gebiet:           INTEGER;
      fenster:          WindowPtr;
      menuErg:          LONGINT;
BEGIN
      gebiet := FindWindow(wo,fenster);
      CASE gebiet OF
            inMenuBar:
                  BEGIN
                  menuErg := MenuSelect(wo);
                  DoCommand(HiWord(menuErg),LoWord(menuErg));
                  END; {inMenuBar}

            inSysWindow:
                  BEGIN
                  SystemClick(dasEreignis,fenster);
                  END; {inSysWindow}
```

```
          inDesk:
                BEGIN
                { ist uninteressant }
                END; {inDesk}
          inContent:
                BEGIN
                IF (fenster <> FrontWindow) THEN
                      SelectWindow(fenster)
                ELSE
                      IF (GetWKind(fenster) = grafKind) THEN
                            BEGIN
                            GlobalToLocal(wo);
                            KlickInFenster(fenster,wo);
                            END;
                END; {inContent}
          inDrag:
                BEGIN
                DragWindow(fenster,wo,dragRect);
                END; {inDrag}
          inGrow:
                BEGIN
                IF (fenster <> FrontWindow) THEN
                      SelectWindow(fenster)
                ELSE
                      BEGIN
                      WindowGrow(fenster,wo,growRect);
                      AenderFenster(fenster);
                      END;
                END; {inGrow}
          inGoAway:
                BEGIN
                IF TrackGoAway(fenster,wo) THEN
                      SchliesseFenster(fenster);
                END; {inGoAway}
          END;{CASE}
END;{Mausklick}


PROCEDURE BearbeiteEreig;
VAR
      ch:                 CHAR;
      menuErg:            LONGINT;
      dialog:             DialogPtr;
      itemHit:            INTEGER;

BEGIN

      SystemTask;
```

```
    IF GetNextEvent(everyEvent,dasEreignis) THEN;
    WITH dasEreignis DO
          BEGIN
          fMouse   := (BitAnd(modifiers,128) <> 0);
          fShift   := (BitAnd(modifiers,512) <> 0);
          fShLock  := (BitAnd(modifiers,1024)<> 0);
          fOption  := (BitAnd(modifiers,2048)<> 0);
          fCommand := (BitAnd(modifiers,256) <> 0);
          fActive  := (BitAnd(modifiers,001) <> 0);
          END;

    IF  IsDialogEvent(dasEreignis)
    AND  NOT(
                (dasEreignis.what = keyDown )
           AND fCommand)
    THEN
          BEGIN
          IF DialogSelect(dasEreignis,dialog,itemHit)
          THEN
                DialogKlick(dialog,itemHit);

          END
    ELSE
    WITH dasEreignis DO
          BEGIN
          CASE what OF
                mouseDown:
                      MausKlick(where);

                autoKey,keyDown:
                      BEGIN
                      ch := CHR(BitAnd(message,255));
                      IF fCommand AND (what = keyDown) THEN
                            BEGIN
                            menuErg := MenuKey(ch);
                            DoCommand(HiWord(menuErg),
                                        LoWord(MenuErg));
                            END;
                      END;

                updateEvt:
                      WindowUpdate(message);

                activateEvt:
                      WindowActivate(message,fActive);

          END;{CASE};
          END;
END;{BearbeiteEreig}
```

```
(* ********************************************
*********************************************

                    INITIALISIERUNG/TERMINIERUNG

*********************************************
******************************************* *)


PROCEDURE InitMac;
BEGIN
  InitMenus;
  dragRect := screenbits.bounds;
  dragRect.top := dragRect.top + 20;
  growRect := dragRect;
  InsetRect(dragRect,4,4);
  growRect.top := 60;
  growRect.left := 100;
END;{InitMac}


FUNCTION CreatePict: PicHandle;
VAR
      pict:             PicHandle;
      saveClip:         RgnHandle;
      r:                Rect;
      str:              Str255;
BEGIN
      GetIndString(str,idStr,1);
      SetRect(r,1,1,50,50);
      SaveClip := NewRgn;
      GetClip(saveClip);
      ClipRect(r);

      pict := OpenPicture(r);
            PenSize(1,1);
            PenPat(black);
            PenMode(patXor);
            TextSize(9);
            TextMode(patXor);
            TextFont(2);
            FrameRect(r);
            SetRect(r,10,10,20,20);
            FrameRect(r);
            PenPat(dkGray);
```

```
            PenSize(1,1);
            SetRect(r,30,10,45,20);
            FrameArc(r,0,250);
            PenPat(ltGray);
            PaintArc(r,0,250);
            MoveTo(5,45);
            DrawString(str);
      ClosePicture;
      CreatePict := pict;

      SetClip(saveClip);
      DisposeRgn(saveClip);
END;


PROCEDURE InitProg;
VAR
      i:          INTEGER;
      bounds:     Rect;

BEGIN

      FOR i:= firstMenu TO lastmenu DO
            menu[i]:= GetMenu(i);
      AddResMenu(menu[mApfel],'DRVR');

      FOR i:= firstMenu TO lastmenu DO
        InsertMenu(menu[i],NIL);
      DrawMenuBar;

      nonModal := GetNewDialog(1001,NIL,POINTER(-1));

      fenster1 := GetNewWindow(idWin1,NIL,
                                          POINTER(-1));
      SetWKind(fenster1,grafKind);

      fenster2 := GetNewWindow(idWin2,NIL,
                                          POINTER(-1));
      SetWKind(fenster2,grafKind);

      AenderFenster(fenster1);
      AenderFenster(fenster2);

      SetPort(fenster1);
      SelectWindow(fenster1);

      wPict := CreatePict;
      scale := 8;
END;{InitProg}
```

```
PROCEDURE BeendeProgramm;
BEGIN
     SetCursor(GetCursor(4)^^);   { Uhr-Cursor }
     WHILE (FrontWindow <> NIL) DO
          SchliesseFenster(FrontWindow);
END;{BeendeProgramm}

(* *******************************************
*********************************************

                              HAUPTPROGRAMM

*********************************************
****************************************** *)


BEGIN {HauptProgramm}
     InitMac;
     InitProg;
     ende := FALSE;
     REPEAT
          BearbeiteEreig;
     UNTIL ende;
     BeendeProgramm;
END.{HauptProgramm}
```

Stichwortverzeichnis

Weitere Fachbücher aus unserem Verlagsprogramm

IBM-PC, XT

100 BASIC-Programme und Subroutinen für den IBM-PC
Mai 1985, 308 Seiten
Diese Sammlung von BASIC-Programmen für den IBM-PC oder XT ist eine Fundgrube an Techniken und Informationen, die sich auch durch Studium der Handbücher nicht erschließen lassen. Viele der erarbeiteten Programme lassen sich vom Leser auch problemlos in eigene Anwendungen einbauen. Besonders bequem: sie sind auch auf Diskette erhältlich.
Best.-Nr. 718, ISBN 3-89090-095-X
(sFr. 63,50/öS 538,20) **DM 69,–**

PEEKs und POKEs für IBM-PCs
Mai 1985, 46 Seiten
Besitzen Sie einen IBM-PC, einen IBM-AT oder einen IBM-Kompatiblen, und interessieren Sie sich für das Innenleben Ihres Rechners? Wollen Sie wissen, wie man gerätespezifische Funktionen optimal in eigene Programme einbindet? Dann ist dieses Buch genau das richtige für Sie! Lesen Sie dieses Buch, lassen Sie die auf der beiliegenden Diskette gespeicherten Programme laufen und Sie erfahren viel Interessantes und Nützliches über Ihren Rechner.
Best.-Nr. MT 617, ISBN 3-89090-127-1
(sFr. 81,–/öS 686,40) **DM 88,–**

PC — Das intelligente Werkzeug für Jedermann
2. überarbeitete Auflage Oktober 1985, 352 Seiten
Was ist und was kann ein Personal Computer · Einsatzgebiete, Aufbau und Funktionsweisen von Personal Computer-Systemen · Zentraleinheit · Tastatur und Bildschirm · Massenspeicher · Schnittstellen · Hardware-Erweiterungen · Mehrbenutzersysteme · Netzwerke · Betriebssysteme · Programmiersprachen im Vergleich · Software woher · Auswahlkriterien · Blick in die Zukunft.
Best.-Nr. MT 791, ISBN 3-89090-115-8
(sFr. 44,20/öS 374,40) **DM 48,–**

So programmiert man 16-Bit-Computer
August 1985, 300 Seiten
»Die besten Lehrmeister sind gute Programmbeispiele«. Unter diesem Motto bringt dieses Buch über 20 Profi-Programme, vom einfachen Spielprogramm in BASIC bis zum umfangreichen Einheitentreiber in Assembler. Der Einstieg wird leicht gemacht durch »Reversi«, einem Brettspiel gegen den Computer. Das Programm »Querverweisliste« für BASIC-Programme erleichtert mit strukturiertem Listing und Variablenlisten die Dokumentation. **Alle Quellprogramme auf Diskette im Buch enthalten.**
Best.-Nr. MT 816, ISBN 3-89090-163-8
(sFr. 68,10/öS 577,20) **DM 74,–**

Die IBM-Personal Computer
2. überarbeitete Auflage
Februar 1985, 350 Seiten
Die IBM-PCs in ihrer Hard- und Software · Betriebssysteme · Programmiersprachen · Textverarbeitung · Tabellen- und Planungsprogramme · zusätzliche Hardware- und Softwareprodukte · IBM-PC-kompatible Rechner und Mitbewerbersysteme.
Best.-Nr. MT 630, ISBN 3-922120-93-8
(sFr. 53,40/öS 452,40) **DM 58,–**

Die Welt des IBM-PC
Juli 1984, 444 Seiten
Ein praktisches Handbuch für den mühelosen Umgang mit Ihrem IBM-PC · ausführlichst erläuterte Befehle · Programmentwicklung in BASIC · Menüs und andere Hilfsmittel · Grafikprozeduren · mit Beispielen auf Diskette (im Buch enthalten!).
Best.-Nr. MT 636, ISBN 3-89090-042-9
(sFr. 81,–/öS 686,40) **DM 88,–**

CAD mit Personal Computer
1. Quartal 1986, ca. 400 Seiten
Ein Überblick über den derzeitigen CAD-Technologiestand · die Low-Cost-CAD-Systeme auf PC-Basis · ein Konstruktionsbeispiel · der ideale Leitfaden zur Durchführung von CAD-Aufgabenstellung!
Best.-Nr. MT 755, ISBN 3-89090-156-5
(sFr. 75,40/öS 639,60) **DM 82,–**

Programmieren mit dem IBM-PC: BASIC
April 1984, 442 Seiten
BASIC-Grundlagen für den IBM-PC · Datentypen, Konstanten, Berechnungen und Ausgabe · Variablen: Zuordnung und Input · Programmentwurf · Fehlersuche · Strings · Sequentielle Dateien · Direktzugriffsdateien · Farbe und Grafik.
Best.-Nr. MT 663, ISBN 3-922120-97-0
(sFr. 53,40/öS 452,40) **DM 58,–**

Programmieren mit dem IBM-PC: Pascal
Mai 1984, 459 Seiten
Eine Einführung in das Programmieren mit IBM Pascal · Anwendung des Pascal-Compilers · die Elemente von Pascal · grundlegende Konstruktionen · Module und Einheiten · zahlreiche Beispielprogramme und Übungen ermöglichen eine schnelle Einarbeitung in Pascal.
Best.-Nr. MT 664, ISBN 3-922120-98-9
(sFr. 53,40/öS 452,40) **DM 58,–**

Programmieren mit dem IBM-PC: UCSD Pascal
September 1984, 537 Seiten
Ein Lehr- und Übungsbuch für das Programmieren in UCSD Pascal mit dem IBM-PC · methodisch aufgebaute Lernabschnitte mit Beispielen aus dem Bereich der Datenverarbeitung · Programme für Berechnungen und Ausgabe · Verarbeitung von Textinformation · Dateien und Records · Datenstrukturen.
Best.-Nr. MT 713, ISBN 3-89090-049-6
(sFr. 53,40/öS 452,40) **DM 58,–**

Die angegebenen Preise sind Ladenpreise



MARKUS BREUER

Jahrgang 1960, ist zur Zeit noch Student der Informatik an der Universität Dortmund. Studienschwerpunkte: Software-Ergonomie, Datenbanken und Objekt-orientierte Systeme. Thema der Diplomarbeit: Entwicklung einer neuartigen, Objekt-orientierten Benutzeroberfläche für Datenbanken und Information-Retrieval-Systeme.

Neben dem Studium langjährige Tätigkeit in der Entwicklung und Realisierung verschiedener Programmsysteme auf Mikrocomputern, in jüngster Zeit vor allem auf Lisa und Macintosh.

Freier Autor mit den Spezialgebieten Programmiersprachen, Software-Entwicklungen, Macintosh und andere hochentwickelte Benutzerschnittstellen.

Die Programmierung des Macintosh

Der Macintosh ist anders als andere Computer! Er hat den Weg für eine ganze Generation von neuen Mikrocomputern bereitet. Diese neue Generation kommt jetzt nach und nach auf den Markt und hat vor allen Dingen eines gemeinsam: Die neuen Computer sind sehr viel einfacher und leichter zu bedienen als alle Computer vorher.

Dieses Buch wendet sich an diejenigen, die selbst Programme entwickeln wollen, die die Eigenschaften des Macintosh voll ausnutzen. Gerade die Eigenschaften, die den Macintosh leicht bedienbar machen, machen ihn aber oft auch schwer programmierbar, weil sie so neu und ungewohnt für den Programmierer sind. Um die Programmierung zu vereinfachen, hat Apple zwar eine Fülle hilfreicher Unterprogramme fest in den Macintosh eingebaut, die sogenannte ToolBox (zu deutsch »Werkzeugkiste«) im ROM des Mac, aber gerade diese Fülle ist am Anfang schwer durchschaubar und macht die Einarbeitung um so schwieriger.

Das Buch gibt einen Überblick über die wichtigsten Gruppen der ToolBox-Routinen und versucht, diese dem Leser schrittweise näherzubringen. Behandelt werden unter anderem:

- Das Grafikpaket QuickDraw (inklusive bewegter Grafiken)
- Die Verwaltung von überlappenden Fenstern
- Das Erstellen eigener PullDown-Menüs
- Die Verwendung von »Schreibtisch-Zubehör«
- Die Programmierung von Eingabe-Masken (Dialogen)

Nach der Erläuterung der wichtigsten Einzel-Pakete der ToolBox wird das Erlernte anhand eines Beispielprogrammes erprobt, das alle wesentlichen Eigenschaften des Macintosh nutzt. Dieses Beispielprogramm ist so konzipiert, daß es als »Gerüst« für die weitere Entwicklung eigener Programme genutzt werden kann. Alle Programm-Beispiele orientieren sich an der Programmiersprache Pascal, die in mehreren Versionen für den Macintosh erhältlich ist. An vielen Stellen werden jedoch auch Hinweise gegeben, wie die Beispiele in andere Programmiersprachen übertragen werden können.

Software-Anforderung: Pascal-Version für den Macintosh

Hardware-Anforderung: Macintosh mit 128 KByte Arbeitsspeicher

ISB N 3-89090-184-0



Markt&Technik
Verlag Aktiengesellschaft

DM 54,-
sFr. 49,70
öS 421,20